Estudio e implementación de reconocedores de secuencias mediante hardware evolutivo

Alberto Urbón Aguado

Director: Dr. Juan Lanchares Dávila

Proyecto de Fin de Máster

Máster en Investigación en Informática

Facultad de Informática
Universidad Complutense de Madrid
2008/2009

Índice de Contenidos

Índice de Ilustraciones

Autorización de Difusión

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "Estudio e implementación de reconocedores de secuencias mediante hardware evolutivo", realizado durante el curso académico 2008-2009 bajo la dirección de Juan Lanchares Dávila en el Departamento de Ingeniería de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Fdo. Alberto Urbón Aguado

En Madrid, a 14 de Septiembre de 2009.

El éxito es aprender a ir de fracaso
en fracaso sin desesperarse.

Sir Winston Churchill.

Agradecimientos

Nadie dijo que fuera a ser corto, ni fácil. Y como no podía ser de otra manera, así no ha sido. En cualquier caso, y tras mucho trabajo, aquí esta el resultado. Con estas líneas quisiera agradecer a las personas que de una u otra forma han colaborado y me han ayudado a sacar adelante este proyecto.

En primer lugar, quería agradecer enormemente a Pablo García del Valle su entrega, ayuda e infinita paciencia al ser capaz de responder a mi aluvión interminable de preguntas a lo largo de este año… ya sea por mail, teléfono o presencialmente… ¡e incluso en pleno agosto en una cafetería de Moncloa! Gracias por compartir tus conocimientos y por tu ayuda para introducirme en el interesante mundo del hardware reconfigurable.

También quería agradecer a Juan Lanchares Dávila su disposición inicial cuando allá por septiembre del año pasado le propuse hacer algún proyecto "trabajando con FPGA y VHDL…" Finalmente, creo que lo hemos conseguido, y a estas páginas me remito. También por su apoyo durante estos meses y por iniciarme en el mundo de la investigación.

Y mi más sentido y sincero agradecimiento, como no podía ser de otra manera, hacia mis padres Félix y Mª Carmen, por su apoyo incondicional a todos mis proyectos durante estos 25 años.

Resumen

El Hardware Evolutivo es un esquema de diseño automático de sistemas hardware inspirado en la naturaleza [3]. Los circuitos son codificados como cromosomas de un algoritmo evolutivo, son posteriormente cargados a hardware reconfigurable y evaluados para obtener su aptitud o fitness, según la cual son clasificados [14]. Como campo de investigación es relativamente reciente, pues nació a principios de los noventa, siendo ya en 1993 cuando se publicaron las primeras investigaciones sobre la materia [1], [2]. Surgió de la conjunción de dos campos claramente diferenciados, por una parte, de la aplicación de las teorías darwinistas provenientes del estudio de la propia Naturaleza en la informática mediante los algoritmos genéticos [36], y por otra, del desarrollo del hardware reconfigurable para fines comerciales, principalmente mediante dispositivos tales como las FPGA (Field Programmable Gate Array). Éstos son, básicamente, chips VLSI (Very Large Scale of Integration) que contienen una gran cantidad de lógica reconfigurable y programable por el usuario [66].

Mediante este trabajo de investigación se ha querido principalmente desarrollar un sistema funcional a través del uso de estas técnicas y teorías. Partiendo de las ideas de otros autores duchos en la materia, ya que el sistema se basa en VRCs (Virtual Reconfigurable Circuit) [14], esta investigación trata el desarrollo y la implementación un sistema secuencial usual como es un reconocedor de patrones. Como característica fundamental, cabe destacar que se trata de un diseño evolutivo, por lo que es el propio sistema el que busca una implementación válida. Ésta búsqueda se realiza a partir de una serie de restricciones y una arquitectura base impuestas de antemano. El sistema es lo suficientemente claro como para que se trate de un primer paso en la materia, pero al mismo tiempo novedoso y vanguardista pues hasta ahora la mayoría de los desarrollos publicados han sido sistemas combinacionales.

La metodología de desarrollo llevada a cabo ha estado siempre guiada por la generalidad, es por ello que nos hayamos decantado por un co-diseño HW-SW. Por una parte, su núcleo esta formado por una serie de módulos hardware que se ejecutan a gran velocidad, aspecto necesario en este tipo de sistemas. Por otra parte, el poder programar a cierto nivel de abstracción nos ha permitido desarrollar el corazón de la evolución en lenguaje C, fácilmente intercambiable y/o modificable sin perder prestaciones, pues es ejecutado en un soft-core de alto rendimiento como es el MicroBlaze de Xilinx [63].

Palabras Clave

Hardware Evolutivo, Computación Evolutiva, Algoritmos Genéticos, Componentes Evolutivos, Diseño Automático, Evolución Intrínseca, Hardware Reconfigurable, FPGA, VRC

Abstract

Evolvable hardware is a circuit automatic design scheme inspired directly by Nature [3]. Circuits are coded as chromosomes of a genetic algorithm which are afterwards uploaded to a reconfigurable logic device and thus assessed in order to obtain its fitness value, by means of which they are classified [14]. It is considered a quite recent researching field, since it was born in the early nineties, and actually the first related publications date of 1993 [1], [2]. The field emerged when some authors combined two clearly different ideas, one coming from genetic algorithms [36], considered as the application in computing of the Darwinist laws, and the other by the striking development of the reconfigurable hardware for commercial purposes, above all by means of FPGAs (*Field Programmable Gate Array*). These devices are, basically, VLSI chips (*Very Large Scale of Integration*) that are made of a huge amount of logic ready to be programmed and reconfigured by the developer [66].

The goal of this research is a hands-on study of the field by the development of a fully functional system, always bearing in mind the use of these techniques and theories. Starting off from the ideas of other experienced authors, the system we have developed is based in VRCs (*Virtual Reconfigurable Circuit*) [14], and it has the functionality of an usual pattern recognizer. As a main feature, it has to be pointed out that it is an evolvable design, i.e. it looks for a valid implementation by itself. This process is carried out assuming a set of constraints and a reconfigurable architecture prepared in advance. The system is clear enough to be considered as a first step in the field, but likewise original and state-of-the-art since up to now almost all systems developed by these techniques have been combinational.

Throughout the research, we have always been focused towards generality and thus, the decision was to choose a Hw-Sw co-design. On the one hand, its core is based on a series of hardware modules which are rapidly executed, a necessary characteristic in this sort of systems. On the other hand, the fact of being able to program at high level allowed us to perform the main part of the evolution in a powerful language as C. That entails an ease in the modifications and improvements, and the avoidance of loosing much performance since this code is executed in a high-performance soft-core as Xilinx MicroBlaze [63].

Keywords

Evolvable Hardware, Genetic Algorithms, Evolvable Computing, Evolvable Components, Automatic Design, Intrinsic Evolution, Reconfigurable Hardware, FPGA, VRC

1. MOTIVACIÓN Y OBJETIVOS

En la actualidad los sistemas electrónicos son cada vez más complejos. Para ser capaz de afrontar el diseño de estos sistemas, los diseñadores utilizan la abstracción, la técnica del divide y vencerás, y un conjunto de poderosas herramientas de diseño automático. Pero estas aproximaciones tienen varias desventajas. La abstracción conlleva la pérdida de información y detalles sobre el circuito a implementar. Las herramientas CAD se basan en rígidas reglas de diseño pensadas para realizar una síntesis sistemática del circuito sin permitir explorar circuitos que, aun no cumpliendo estas reglas, presentan un mejor comportamiento. Y, finalmente, la técnica de divide y vencerás prefija el comportamiento de los diferentes módulos al determinar cómo se van a relacionar estos módulos entre sí, perdiendo la oportunidad de, al permitir relaciones más flexibles, encontrar diseños más eficientes.

Cuanto más complejo es un circuito, más probable es que aparezcan fallos y errores. Según [5], existe un tipo de sistemas llamados sistemas no sostenibles, en los cuales no es posible cambiar o reparar los dispositivos dañados, o bien por ser excesivamente costoso el proceso o bien por ser imposible. Ejemplos de este tipo de

sistemas son los satélites, las pruebas de alta profundidad en el mar o las misiones espaciales de larga duración.

Cuando aparece un fallo en un sistema normal, éste puede ser detectado y solucionado simplemente cambiando el dispositivo dañado. Además, existen sistemas que son tolerantes a fallos, es decir pueden seguir trabajando aunque alguno de los circuitos este dañado. Generalmente esta tolerancia se consigue mediante redundancia HW: si un circuito falla, existen otros circuitos en el sistema que implementan la misma funcionalidad y por lo tanto el sistema es capaz de seguir trabajando hasta que el dispositivo es reparado o cambiado.

Esta solución no es aplicable en los sistemas no sostenibles. Cuando en estos sistemas aparece un fallo, este puede ocultarse mediante redundancia pero este dispositivo no vuelve a funcionar. Si la misión es de larga duración, puede que todos los dispositivos fallen y por lo tanto la misión en su totalidad acabe fracasando. Este caso se podría dar en una misión espacial pensada para durar 50 o 100 años. Durante este tiempo es muy probable que los circuitos tengan fallos.

El Hardware Evolutivo (Evolvable Hardware –EHW-) une el HW reconfigurable, la inteligencia artificial, la tolerancia a fallos y los sistemas autónomos para intentar solucionar algunos de estos problemas. Básicamente consiste en un HW que es capaz de modificar su comportamiento autónomamente, es decir, sin intervención externa, para adaptarse a la funcionalidad requerida. Esta adaptación se realiza en tiempo de ejecución y es debida o bien a que las condiciones del entorno han cambiado (temperatura, radiaciones sobre el circuito…) y el circuito original ha dejado de funcionar correctamente o bien a que se ha producido un fallo en el alguno de los componentes del sistema. En este último caso, el EHW tendría que ser capaz de detectar la parte dañada del circuito y de recuperar la funcionalidad inicial, esquivando las partes del circuito que no funcionaran.

El HWE se inspira en la evolución de las especies. Por lo tanto, trabaja sobre poblaciones de circuitos cada uno de los cuales es una posible solución al problema, es decir, es un circuito que implementa el comportamiento requerido. Para simular la evolución se seleccionan los circuitos que más se aproximan al comportamiento deseado. Esta proximidad se mide evaluando la función de coste. A continuación, y para obtener la siguiente generación de circuitos, mediante una serie de operadores genéticos se intercambia información entre parejas de circuitos padres para generar circuitos hijos. Este proceso se repite hasta que se obtiene el circuito que cumple totalmente las especificaciones deseadas.

El EHW usa algoritmos bio-inspirados (algoritmos genéticos, programación evolutiva o programación genética) para realizar la evolución. Finalmente, el circuito obtenido se implementa sobre una FPGA.

Las características del EHW pueden ayudar a solucionar tanto los problemas de diseño de circuitos como los problemas de tolerancias a fallos y adaptación a entornos cambiantes. En el primer caso porque el EHW no usa unas reglas de diseño rígidas, en realidad no usa reglas de diseño, solo se guía por el comportamiento del circuito y por lo tanto el espacio de soluciones sobre el que trabaja es mayor. En el

segundo caso porque permite que el sistema se rediseñe de manera autónoma, bien cuando aparece un fallo, bien cuando las características del entorno varían.

Aun siendo un área de conocimiento relativamente nueva (los primeros artículos que hacen referencia a ella son los trabajos [1], [2], [50] y [51] de principios de lo 90), tanto en el campo del diseño de circuitos [52], [53], [48], [49], [39], [40], [43], [45], [46], [47], [41] y [42] como en el de sistemas auto-adaptativos on-line [33], [54], [30], [25], [26] y [55] existe una gran diversidad de trabajos publicados, principalmente en la *International Conference On Evolvable HW* y *La NASA-DOD Evolvable HW Conference* [7].

A la hora de emprender este trabajo nos pareció interesante realizar una primera aproximación al estudio e implementación de un sistema EHW que fuera capaz de evolucionar de manera autónoma según fuera el comportamiento del entorno.

Visto lo anterior, el objetivo de este trabajo es el estudio de un sistema auto-adaptativo on-line basado en las técnicas de Hardware Evolutivo. La principal característica de esta aproximación es que el HW no es estático sino que es capaz de adaptarse, sin intervención exterior, tanto a los cambios del entorno de trabajo, como recuperarse de manera autónoma de posibles fallos en el HW de manera que puede seguir cumpliendo la misión para la que fue diseñado. Concretando, se va a implementar un circuito evolutivo capaz de reconocer varias secuencias diferentes de ceros y unos en función del valor del entorno. En este caso, este entorno se simula mediante unos interruptores que representan el valor de la secuencia a reconocer.

En una aproximación clásica, el sistema tiene almacenado cada uno de los circuitos necesarios para implementar cada uno de los reconocedores de secuencias. Cuando la señal que simula el entorno indica que éste ha cambiado, se selecciona el reconocedor adecuado y se carga en la FPGA.

En nuestra aproximación, cuando se detecta un cambio en el entorno, no se carga directamente el nuevo circuito, sino que, aplicando técnicas evolutivas, el HW evoluciona hasta acabar implementado el circuito deseado. Es decir, los diferentes posibles circuitos no están almacenados en el sistema, sino que se generan de forma autónoma. Esto puede ser de gran utilidad en cierto tipo de sistemas, como los utilizados en misiones espaciales de larga duración, en los que se ignora cómo se va a comportar el entorno y por lo tanto no es posible almacenar todos los posibles circuitos que pudieran necesitarse.

Como la idea es estudiar el EHW en entornos cuyo comportamiento se desconoce las características que debería tener este sistema son:

- La evolución debe ser on-line, es decir no se realizará mediante simulaciones externas y luego se cargará su resultado sobre la FPGA, sino que se llevará a cabo sobre el mismo HW que posteriormente implementará el circuito. De esta manera se evita el problema de la portabilidad.

- La evolución será intrínseca, es decir las posibles soluciones de la población se evaluarán directamente sobre la FPGA.
- La evolución será sin restricciones. Es decir, como no se conoce la posible evolución del entorno, las restricciones pueden provocar que no se alcance el circuito deseado.
- Se utilizará la técnica de los componentes evolutivos, cuyo objetivo es generalizar el EHW. Es decir, el cálculo de la función de fitness no está incluida en el propio módulo evolutivo, de esta forma el EHW puede utilizarse en diferentes aproximaciones.
- El sistema tiene que ser tolerante a fallos, es decir si alguno de los módulos que componen un reconocedor falla, el sistema tiene que ser capaz de recuperarse por si mismo de este fallo.

2. TEORIA DEL HARDWARE EVOLUTIVO

Como ya se ha visto en las motivaciones, el Hardware Evolutivo es un sistema de diseño automático de hardware inspirado en la naturaleza [3]. Los circuitos se codifican como cromosomas de un algoritmo evolutivo, posteriormente son implementados mediante hardware reconfigurable y evaluados para obtener su aptitud, según la cual son clasificados [14]. En este capítulo se estudian las bases teóricas del EHW. En primer lugar se dan diferentes clasificaciones EHW, a continuación se realiza un estudio de la forma del espacio de soluciones, de la función de coste para circuitos integrados, la forma de evaluar los circuitos y la forma más habitual de representar el problema. También se estudian los componentes evolutivos, que es una aproximación que intenta diseñar HW genético genérico. Por último, se estudian las dos principales utilidades de este tipo de HW: el diseño evolutivo, y los sistemas auto-adaptativos on-line. Por último, se hace hincapié en el principal escollo que posee este enfoque, el problema de la escalabilidad.

2.1. PRINCIPALES CARACTERÍSTICAS DEL EHW.

Algunas de las principales características que debe tener el EHW son [5]:

- El circuito final se obtiene mediante evolución y no utilizando técnicas clásicas de síntesis y diseño. El diseñador sólo trabaja con el comportamiento del circuito, es decir, se preocupa de *qué* se quiere hacer, no de *cómo* se quiere hacer.
- En el EHW, no se pude hablar de un circuito óptimo sino de un circuito que trabaja correctamente.
- Este tipo de circuitos tienen una tolerancia a fallos natural. El EHW trabaja sobre una población de diferentes circuitos, y como cada circuito es distinto, compite con el resto para sobrevivir y reproducirse. Cuando aparece un fallo (tanto interno como externo) este fallo no tiene porque afectar de la misma manera a todos los circuitos de la población. Y podría suceder que algunas de los circuitos de la población no se vieran afectados por él.
- Este tipo de sistemas es capaz de adaptarse a variaciones en el entorno, aunque como veremos, esto depende de cuándo se pare la evolución.
- Al ser circuitos que no se diseñan bajo reglas, la mayoría de las veces no son verificables. Es decir, es muy difícil analizar el sistema para conocer como se hace el trabajo.

2.2. LA ESTRATEGIA EVOLUTIVA.

Uno de de los conceptos clave que definen inequívocamente el hardware evolutivo es la estrategia de evolución implementada [36], [27]. Ésta se inspira directamente de la teoría de la evolución darwinista, la cual plantea que durante la evolución de las especies en la propia Naturaleza se produce en cada generación una selección de los mejores individuos, los cuales a su vez son los procreadores y progenitores de la siguiente generación. Abstrayendo estos conceptos, y suponiendo que cada individuo representa un circuito electrónico, caracterizamos cada uno de ellos mediante un array de bits que inequívocamente representan las conexiones, funciones y demás elementos constituyentes del circuito. Así, cada uno de los individuos es evaluado y clasificado según una función de aptitud o *fitness* arbitraria, tras lo cual se lleva a cabo la selección, dónde sólo los mejores sobreviven pasando a la siguiente generación. Éstos serán los progenitores de una nueva población, concebida mediante la variación de sus individuos. Ésta representa un proceso aleatorio por el cual se conciben nuevos cromosomas a partir de los existentes cambiando alguno de

los parámetros de estos últimos. Entre los operadores de variación más comunes destacan:

- Mutación. Se lleva a cabo con un solo progenitor y supone la creación de un nuevo individuo mediante el intercambio aleatorio del valor de una posición concreta.

- Cruce. a partir del material genético de dos o más progenitores, se desarrolla un nuevo individuo. En los arrays binarios, esto supone intercambiar ciertos subarrays de ambos padres.

Partiendo de una representación en forma de cadena binaria de los cromosomas según la CGP (Cartesian Genetic Programming), son los algoritmos evolutivos los encargados de recorrer el espacio de soluciones factibles. Éstos han venido desarrollándose interrumpidamente desde finales de los años 60 del siglo pasado [27]. Desde que nacieron han surgido tres paradigmas básicos, son:

- Programación Evolutiva. Este paradigma se caracteriza porque los individuos quedan inequívocamente representados como ternas cuyos valores representan estados de un autómata finito. Sus valores son el estado actual, el siguiente y un símbolo del alfabeto utilizado. Encontrándonos en un estado, si el símbolo que tenemos es el mismo con el que cuenta ese estado, se produciría una transición al siguiente estado.

- Estrategias Evolutivas. En vez de cadenas, este paradigma trabaja con una población de individuos pertenecientes al dominio de los números reales. Partiendo de un único operador de variación, la mutación, los individuos evolucionan hasta alcanzar el óptimo de la función objetivo.

- Algoritmos Genéticos. Presentan las posibles soluciones, o genotipo, como cadenas binarias cuyo formato tiene un significado concreto, fenotipo. Partiendo de una población inicial aleatoria de individuos, éstos van siendo evaluados por una función de aptitud que los clasifica. Posteriormente son seleccionados sólo los mejores de cada generación, técnica conocida como elitismo. De cara a obtener una nueva generación se parte de los operadores de variación de cruce y mutación. En la Ilustración 1 podemos ver cómo se podría implementar.

```
Initialize population of random alleles
REPEAT
        Evaluate individuals
        Select individuals into 'mating pool'
        REPEAT
                Randomly take an arbitrary number of parents from the 'mating pool'
                Use random crossover and mutation to generate offspring
                Place offspring into population
        UNTIL population is filled with new offspring
UNTIL termination condition is satisfied
```

Ilustración 1. Pseudo-código de un algoritmo genético.

2.3. DIFERENTES CLASIFICACIONES DEL EHW.

En la literatura sobre EHW se puede encontrar un elevado número de clasificaciones en función de diferentes parámetros. En este trabajo solo se estudiaran aquellas clasificaciones relacionadas directamente con el estudio propuesto. A saber:

- Evolución on-line vs. evolución off-line.
- Evolución intrínseca vs. evolución extrínseca.
- Evolución con ligaduras vs. sin ligaduras.

En los siguientes epígrafes se detalla cada una de las clasificaciones.

2.3.1. EVOLUCIÓN ON-LINE VS. EVOLUCIÓN OFF-LINE.

La evolución de un circuito se puede realizar en tiempo de ejecución, es decir mientras que el circuito está trabajando activamente o antes de que el circuito sea implementado. El primer caso se llama **evolución on-line** mientras que al segundo caso se le llama **evolución off-line**. Según cuál sea el objetivo del EHW se deberá utilizar un tipo u otro de evolución.

En el caso del diseño de circuitos, lo importante es obtener un diseño que implemente correctamente las especificaciones. La evolución se realiza únicamente durante la fase de diseño, pero una vez que el circuito está implementado no evoluciona. Suele existir interacción con el diseñador. Para obtener este circuito se

suelen usar todos los recursos de un laboratorio de diseño. Además el ciclo de diseño puede ser relativamente largo, siempre que no sobrepase los límites establecidos por el tiempo de mercado. En este caso se usa evolución off-line.

En el caso de HW adaptativo, la evolución se debe realizar siempre en tiempo de ejecución, luego será por defecto on-line. Hay que indicar que la principal diferencia con el caso anterior, en lo que a recursos se refiere, es que existen limitaciones, tanto de potencia de cálculo, como de memoria. El tiempo necesario para alcanzar el circuito suele ser limitado, no existe el instrumental disponible en un laboratorio, no hay interacción con el diseñador y en muchas ocasiones se realiza sin conocimientos previos de cómo va a evolucionar el entorno.

2.3.2. EVOLUCIÓN INTRÍNSECA VS. EVOLUCIÓN EXTRÍNSECA.

Existen dos formas de generar el valor de la función de coste de un circuito: extrínsecamente e intrínsecamente. En el primer caso se usa un simulador para evaluar los circuitos y, generalmente, solo se implementa en la FPGA el circuito final obtenido tras la evolución. En el segundo caso todos los circuitos se evalúan mediante HW implementándolos en la FPGA [1], [6].

El tipo de evolución influye sobre el número de veces que la FPGA va a ser reconfigurada, por lo tanto en el tiempo que se tarda en alcanzar el mejor circuito. En la evaluación extrínseca la FPGA solo se configura una vez, mientras que en la intrínseca se reconfigura para cada circuito. En general se puede decir que la evaluación en HW es más rápida y precisa, pero no se debe olvidar que el tiempo de reconfiguración puede ser demasiado grande para algunos sistemas. Por ejemplo en aquellos sistemas que necesitan recuperarse de un fallo en un tiempo mínimo [8], [23].

2.3.3. EVOLUCIÓN SIN LIGADURAS VS. EVOLUCIÓN CON LIGADURAS.

Se ha demostrado [57], [58] que los resultados obtenidos en la evolución pueden ser diferentes dependiendo de si la evolución es extrínseca o intrínseca. Incluso la evolución sobre dos FPGAS idénticas del mismo fabricante puede dar lugar a circuitos diferentes que se comportan de maneras distintas según variaciones del entorno (temperatura, picos de potencia, radiaciones…) A este hecho se le llama el **problema de la portabilidad** [59].

Este problema aparece porque la evolución se realiza **sin ligaduras**, es decir la función de coste solo incluye el comportamiento que se desea para el circuito, pero no se añade a ella ninguna información más. En estos casos, de manera implícita la

evolución tiene en cuenta factores no incluidos en la función de coste. Por ejemplo, si realizas la evaluación extrínsecamente, hay que saber que los simuladores no son tan precisos como la evaluación sobre el propio HW. Como la información que utiliza la evolución para evaluar los circuitos es la proporcionada por el simulador el diseñador se puede encontrar con la sorpresa de que al implementar el circuito en HW, este no funcione.

En el caso de la evolución sobre dos FPGA similares, también pude ocurrir lo mismo puesto que pueden entrar en juego factores como las variaciones de fabricación, o realizar la evaluación en entornos que no van a ser los mismos que en los que va a funcionar finalmente el sistema. Por lo tanto, la evolución sin ligaduras es capaz de explotar las características físicas de un chip y las características del entorno. Este tipo de evolución es muy útil para sistemas auto-adaptativos on-line, en los que se desconoce cómo va a evolucionar el entorno y en los que es necesario sacar el máximo provecho al circuito sobre el que se implementan. Sin embargo, no es nada recomendable para el diseño evolutivo, puesto que en estos casos una vez evolucionado el HW se implementará en un entorno que no suele tener nada que ver con el de un laboratorio y por lo tanto el circuito obtenido como resultado de la evolución puede no funcionar correctamente cuando se implementa.

En estos casos se utiliza la **evolución con ligaduras**, que consiste en añadir a la función de coste las restricciones del entorno de trabajo. La evolución con ligaduras obtiene el mismo resultado en todos los casos por lo tanto se elimina el problema de la portabilidad [14].

Por último, Stoica en [59] propuso la evolución mix-trínseca para eliminar el problema de portabilidad. Esta evolución consiste en realizar una mezcla de evaluación intrínseca y extrínseca.

2.4. ESTUDIO DEL ESPACIO DE SOLUCIONES EN EL EHW (LANDSCAPE).

Se ha indicando con anterioridad que la forma de hacer evolucionar la población de circuitos es seleccionar los más aptos y aplicarles ciertos operadores genéticos para obtener la siguiente generación. Para poder comprender como afectan los diferentes operadores genéticos en la evolución de los circuitos es conveniente tener conocimientos sobre la forma del espacio de soluciones de las funciones de coste de los circuitos digitales (a partir de ahora landscape).

Formalmente el landscape es un grafo cuyos vértices son genotipos etiquetados con el valor de su función de coste y cuyas conexiones se definen mediante operadores genéticos. La relación de vecindad se define como la capacidad

de generar un nuevo individuo con un valor de la función de coste similar aplicando un determinado operador genético. El landscape depende de la representación, de la relación de vecindad y de la función de coste. En general, los landscape se clasifican como accidentados, suaves y neutrales. La búsqueda suele ser más sencilla cuando el paisaje es suave, puesto que los operadores genéticos no afectan en gran medida a las soluciones. El paisaje es neutral cuando existen varias soluciones con el mismo valor. Como conocer la forma exacta del landscape no es posible, se suele realizar un estudio estadístico mediante muestreo aleatorio[14]

En [37], [19] se puede encontrar un estudio sobre el landscape de circuitos integrados. La primera conclusión del trabajo es que cuando se usa la programación genética cartesiana para representar el problema, las relaciones de dependencia entre los genes son muy complejas y hacen el análisis muy difícil. Más concretamente tienen un tipo de relación llamada *epistasy* que, según [60], consiste en que dentro del cromosoma no existe una región concreta para fijar un determinado comportamiento del circuito, sino que el comportamiento final se debe a la relación entre diferentes regiones que por sí solas no aportan ninguna información sobre cómo será el circuito final.

El landscape depende de tres factores: la funcionalidad de las puertas lógicas que se usan como base para construir el circuito, el tipo de conectividad que se permite y la conectividad de la salida. Para facilitar el estudio los autores realizaron estudios parciales sobre cada uno de los tres factores, llegando a las siguientes conclusiones:

- La forma del espacio de soluciones debida a las conexiones internas del circuito es la más accidentada.
- La forma del espacio de soluciones debida a las conexiones de salida es la menos accidentada.
- La forma del espacio de soluciones debida al tipo de puertas lógicas usadas para construir el circuito se encuentra entre ambos.

En general las características landscape son:

- Existen diferentes mesetas claramente diferenciadas entre sí.
- Existe mucha neutralidad (se explica más adelante). Esto es importante porque la neutralidad puede evitar que se caiga en mínimos locales, puede ayudar a mejorar los circuitos, y protege a los circuitos de mutaciones perjudiciales.
- La continuidad del espacio de soluciones está relacionada con el tamaño de los circuitos que evolucionan y con la elección de las puertas básicas de construcción de los circuitos.

Las más importantes conclusiones que se pueden extraer del estudio del landscape son:

- El cruce es un operador ineficiente para el EHW [19]. En cambio, al ser el efecto de la mutación más suave que el del cruce, es el operador perfecto para evolucionar.
- El uso de la mutación como operador facilita implementar sistemas tolerantes a fallos, puesto que el propio fallo se puede interpretar como una mutación fallida, o externa al proceso de evolución.
- La evolución es más sencilla si el número de puertas disponible para la evolución es mayor que el número de puertas necesarias para implementar el circuito.

2.5. EVALUACIÓN DEL CIRCUITO: LA FUNCIÓN DE COSTE.

Como se ha indicado varias veces a lo largo del trabajo, lo importante en el EHW no es tanto cómo se va a implementar un circuito, sino cuál es el comportamiento que se desea, y es precisamente este comportamiento el que se fija a través de la función de coste que sirve para evaluar el circuito. A diferencia de lo que ocurre en el diseño clásico, en el que se aplican una serie de reglas bien estudiadas y establecidas para obtener el circuito final, en el EHW se permite que los circuitos evolucionen sin ninguna norma, libremente. La función de coste que describe el comportamiento deseado es la que guía esta evolución pero sin prefijar ningún tipo de estructura.

En su forma más sencilla, la función de coste se puede ver como una tabla de verdad que describe el comportamiento del circuito, de manera que cada posible entrada tiene establecida su salida. La forma de evaluar es simple, sólo hay que calcular el número de salidas generadas por el circuito que se está evaluando que son correctas. Evidentemente el mejor circuito evaluado es aquel en el que todas las salidas generadas son correctas.

Esta técnica se puede aplicar a circuitos muy sencillos, con tablas de verdad de dimensiones reducidas. Pero, ¿qué hacer cuando la complejidad del circuito provoca que las tablas de verdad sean excesivamente grandes? En estos casos el cálculo de todas las posibles entradas supondría un esfuerzo computacional tremendo. Para estos casos se utiliza un conjunto de entrenamiento y un conjunto de test. El conjunto de entrenamiento es un conjunto de entradas con sus salidas correspondientes, que se supone son capaces de capturar el comportamiento. Con este conjunto se hace evolucionar el circuito. Una vez obtenido el circuito final se utiliza el conjunto de test para comprobar que el comportamiento es correcto.

Es importante recordar que la evolución sólo utiliza el comportamiento para realizar la búsqueda de las soluciones. Por lo tanto, el diseñador tiene que asegurase de que el conjunto de entrenamiento es lo suficientemente amplio o robusto como para ser capaz de capturar el comportamiento del circuito bajo cualquier entrada. En cualquier caso, esta característica no se puede asegurar.

En ocasiones el diseñador puede desear capturar otras características del sistema además del comportamiento. Por ejemplo, el entorno conocido en el que va a trabajar o la biblioteca de componentes disponibles para implementar el circuito. En estos casos, esta información se puede añadir a la función de coste, realizando una evolución con ligaduras [58].

2.6. LA PROGRAMACIÓN GENÉTICA CARTESIANA.

2.6.1. LA REPRESENTACIÓN DEL PROBLEMA.

A la hora de codificar los cromosomas como circuitos, hemos optado por la CGP. Éstas son las siglas de *Cartesian Genetic Programming* [35], una forma de programación genética surgida a principios de los noventa y a través de la cual un programa queda representado como un grafo indexado. Éste se codifica como una cadena de enteros, y entradas, salidas y funciones son numeradas secuencialmente. En el caso del hardware evolutivo, suele plantearse la representación del cromosoma como cadenas binarias.

Al hablar de cromosoma se debe hacer una distinción entre el genotipo y el fenotipo. Volviendo a los orígenes de toda esta nomenclatura, las teorías biológicas, el genotipo viene a ser el contenido genético de un individuo. Éste sumado a las condiciones y comportamientos inducidos del ambiente codifican el fenotipo del propio individuo. De esta manera, podemos representar una ecuación conceptual de la forma: Ambiente + Genotipo = Fenotipo. Salvando las distancias, estas ideas se plasman directamente en el hardware evolutivo. Utilizando un símil informático, el genotipo vendría a representar lo que es una clase en la programación orientada a objetos, es decir, cómo es un objeto. En nuestro caso, el objeto sería un array de bits y el genotipo representa su formato o significado de cada *bit*. El fenotipo, sin embargo, representa una instancia de esa clase, es decir, la descripción de un circuito concreto [36]. Para el genotipo, se define un espacio de búsqueda entre todos los posibles formatos que éstos pueden tener. Paralelamente, el fenotipo define el espacio de soluciones para un genotipo concreto. La Ilustración 23 muestra la caracterización de estas técnicas aplicadas sobre un cromosoma binario, que es básicamente el enfoque seguido en este proyecto de investigación.

El genotipo en la CGP representa un array de celdas o nodos rectangular, que en el hardware evolutivo vienen a ser meras puertas lógicas. Si nos adentramos un poco en las consecuencias del enfoque de la CGP, hay una serie de puntos clave que son cruciales para la obtención de futuros resultados, pues van a marcar las pautas de la evolución de los cromosomas. Así, los genotipos quedan entonces representados como cadenas de longitud fija, y debido a que va a haber genes que no representan nada en algunos casos, los fenotipos serán de longitud variable. Esto provoca que haya varios genotipos que inducen el mismo fenotipo y que por ello consiguen el mismo fitness. Según [35] y según estudiamos en el siguiente epígrafe, esto conlleva una neutralidad que resulta necesaria en el proceso de evolución de los cromosomas, pues mejora sustancialmente la búsqueda de buenas soluciones. Lo mismo ocurre con la redundancia que causan los genotipos, ya que puede haber varios genotipos similares teniendo en cuenta que puede haber nodos que no formen parte del grafo que representa el programa, o funciones que pueden ser implementadas de varias formas. Entonces, a mayor redundancia, mejoras en la búsqueda de soluciones factibles. Estas ideas representan en la futura solución la existencia de sub-circuitos que inducen las mismas soluciones y que podrán mejorar la eficiencia de la solución [18].

Profundizando más en la teoría de la Programación Genética Cartesiana, podemos definir que el circuito reconfigurable es modelado como un array de nodos programables de NC columnas y NR filas en el que:

- Un nodo tiene NN entradas y una salida.
- El número de entradas al circuito es NI.
- En número de salidas del circuito es NO.
- Cada entrada a un nodo puede conectarse a algún nodo de salida de una columna anterior o a alguno de las entradas del circuito.
- Cada nodo puede programarse para implementar alguna de las NF posibles funciones, definidas en el conjunto F.
- La conectividad del circuito se define por el parámetro de niveles L, que determina cuantas columnas hacia atrás se puede conectar la entrada de un nodo.
- Los nodos de la misma columna se pueden conectar entre sí.
- Los nodos pueden estar conectados o desconectados.

La longitud del cromosoma viene dada por la expresión $NR \cdot NC(NN+1)+NO$. Cada nodo necesita NN+1 genes para describir su programación, estos genes representan las entradas del nodo y la función que implementa, como el circuito tiene NR*NC el total de genes es NR*NC(NN+1). Además hay que añadirle la información sobre las salidas (+NO).

En el cromosoma la información aparece por columnas. Para numerar los nodos se actúa de la siguiente manera: primero se numeran las entradas del circuito, después se numeran los nodos por columnas y por último se numeran las salidas del sistema.

Si un nodo no está conectado, sus genes son 000.0, siendo los primeros 0 las entradas al nodo y la función representada 0 también.

Es muy importante recordar que no todas las configuraciones que se alcanzan en la evolución son legales, y esto puede llevar a dañar el circuito cuando se cargan. Por lo tanto, deben estudiar con cuidado la representación y poner ligaduras a las mutaciones para asegurarse que los circuitos que se obtienen son circuitos implementables en la FPGA.

2.6.2. REDUNDANCIA Y NEUTRALIDAD.

Debido a la representación elegida, aunque los cromosomas tienen longitud fija, los circuitos que codifican pueden tener un número variable de nodos. También es posible que dos circuitos diferentes tengan el mismo valor de la función de coste. A los cromosomas que dan lugar a esta última situación se les llama neutrales el uno con respecto al otro. La neutralidad es consecuencia de la redundancia. Existen varios tipos de redundancia:

- Redundancia de nodos, que se debe a los nodos que no están conectados (representados en el cromosoma mediante genes llamados inactivos 00..0).
- Redundancia funcional: un sub-circuito puede implementarse con un menor número de nodos. Es decir un subconjunto de nodos puede substituirse por otro subconjunto de nodos que implementan la misma función lógica.
- Redundancia de entrada, un nodo no tienen todas sus entradas conectadas.

Los genes activos, es decir aquellos que representan nodos que están conectados, son los responsables de la redundancia funcional y pueden ser vistos como intrones. Los intrones son código en un individuo de programación genética que no afecta al comportamiento del individuo. Los intrones aparecen por la evolución de estructuras de longitud variable y no afectan directamente a la supervivencia de un individuo [38]. En la programación genética la aparición de intrones no es buena, pero en la programación genética cartesiana se ha demostrado que los circuitos solo incrementan de tamaño si esto es necesario para mejorar el valor de la función de coste [58]. Basándonos en todo lo anterior se puede afirmar que en los experimentos que llevemos a cabo, debemos asegurar que siempre se deben tener más nodos de los necesarios para implementar el circuito, ya que este hecho favorece la neutralidad que su vez facilita la evolución.

2.7. EL PROBLEMA DE LA ESCALABILIDAD EN EL EHW.

Probablemente el mayor problema del EHW es la escalabilidad. Hasta el momento se ha conseguido implementar circuitos de extrema sencillez si se les compara con el grado de complejidad existente en la actualidad. Esto se debe a varios factores que se explican a continuación. El primero de ellos probablemente sea la escalabilidad de la representación [14]. Los circuitos complejos necesitan muchas puertas y conexiones para implementarse. Para poder representar estos circuitos se necesitan cromosomas extremadamente grandes que a su vez dan lugar a espacios de soluciones enormes. Por ejemplo, para implementar un circuito de aproximadamente cien puertas lógicas se necesita un cromosoma de aproximadamente dos mil bits [8].

En [14] y [37]se proponen dos aproximaciones para resolver el problema:

- El nivel funcional. No se utilizan puertas lógicas como elementos programables sino funciones más complejas. Estas funciones se tienen que definir antes de que se ejecute el algoritmo. El problema radica en la manera de seleccionar estas funciones complejas [10].

- La evolución incremental. Esta solución fue definida por Torrensen en [12]y [13] como una estrategia de divide y vencerás. En este caso, las funciones complejas se obtienen mediante evolución también y después se usan como funciones base para conseguir evolucionar el sistema complejo. La principal ventaja de esta aproximación es que el espacio de búsqueda es más sencillo y pequeño que el usado en la evolución normal. El problema de esta aproximación radica en saber cuáles son los sub-circuitos hacia los que queremos evolucionar, es decir, definir las sub-funciones de coste. Existen dos formas de poder definir esta sub-funciones. La primera consiste en considerar bloques aislados que evolucionan, sin considerar que finalmente van a estar formando parte de un sistema más complejo. Esto podría conseguirse evolucionando aisladamente cada una de las salidas. La segunda forma consiste en identificar subconjuntos y hacerlos evolucionar separadamente, pero utilizado conjuntos de entrenamiento completos, no parciales.

2.8. LAS FPGAS.

El concepto de hardware evolutivo viene íntimamente ligado al de hardware reconfigurable, pues es necesario contar con un dispositivo de dichas características en el que descargar el sistema, evaluar su comportamiento y proceder a su reconfiguración dinámicamente en caso necesario. El hardware reconfigurable lo encontramos en la forma de PLDs (*Programmable Logic Device*), que son dispositivos que incorporan multitud de celdas lógicas básicas y una red de interconexión que las une. Siendo programables ambas partes o una de ellas, dependiendo del tipo, es por ello posible definir con ellas una funcionalidad concreta a voluntad del desarrollador. De todas sus variantes, el mayor exponente lo forman las FPGA (Ilustración 2). Por su extensión, facilidad de uso y por su amplio reconocimiento, las hemos escogido como dispositivo de trabajo. Como podemos ver en [65], estos dispositivos están compuestos por una matriz de CLB (*Configurable Logic Block*) y una red de interconexión programables, por lo que resulta ser un dispositivo idóneo para nuestro objetivo. Dependiendo del fabricante y del modelo, la FPGA en cuestión tendrá unas características concretas, como el número de CLBs, el tamaño de la RAM disponible o el modo de configuración. En cualquier caso, nos abstraeremos de estos detalles a la hora de evolucionar diferentes soluciones, pues será la herramienta de síntesis la que se encargue de ellos.

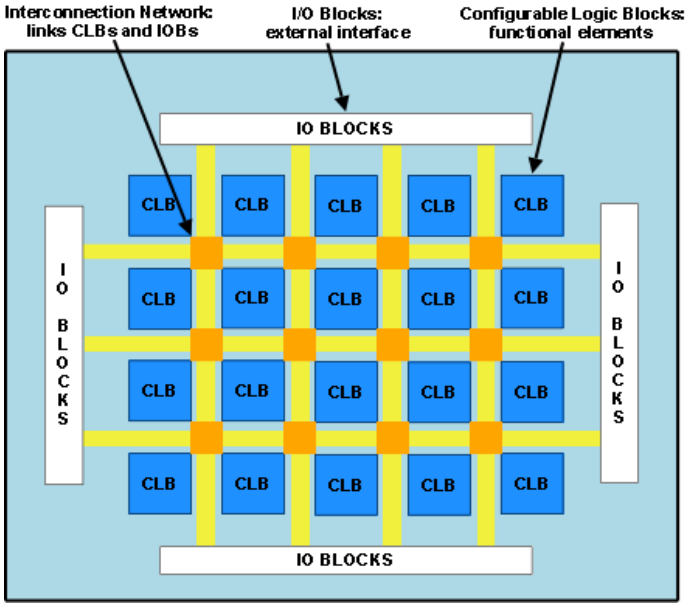

Ilustración 2. Esquema modular genérico de la composición de una FPGA.

2.9. LOS COMPONENTES EVOLUTIVOS.

En sus trabajos [14], [15], [16] y [17] Sekanina propone una nueva aproximación al problema del HWE. Hasta el momento, todos los estudios iban dirigidos a implementar EHW orientados a una aplicación específica, integrando en el módulo genético conocimientos del problema, de manera que el módulo perdía generalidad. El autor propone una metodología para el diseño sistemático de EHW generalista. Para ello supone el sistema dividido en dos partes, el componente evolutivo y el contexto o entorno. Se puede ver el componente evolutivo como una caja negra con sus interfaces perfectamente establecidas. La misión del componente evolutivo es implementar la evolución de la población. La principal diferencia con otros trabajos es que la función de coste no se encuentra en el interior del componente evolutivo sino en lo que llama el contexto, y de esta forma este componente es genérico y por lo tanto puede usarse para solucionar diferentes problemas. En cualquier caso, Sekanina, basándose en el teorema de *no free lunch* (NFL), que plantea que *en promedio todos los algoritmos funcionan igual de bien para todos los problemas y por lo tanto no existe un algoritmo universal que funcione bien para todos los problemas*, se encarga de recordar que este componente evolutivo no es totalmente genérico. Sólo lo es parcialmente y que para que sea óptimo tendría que contener conocimiento suplementario del dominio del problema, lo que le hace perder dicha generalidad. Por lo tanto este modulo genérico en realidad solo es válido para una determinada clase de problemas.

El componente evolutivo se compone de:

- Un circuito reconfigurable.
- Una unidad genética.
- Un controlador dividido.

La unidad genética es el módulo encargado de generar un nuevo cromosoma candidato, estando la búsqueda dirigida por los valores de la función de coste que proporciona el contexto. Esta unidad genética está formada por:

- Operadores genéticos específicos para una determinada clase de problemas.
- Una memoria que almacena los valores de las funciones de coste de las soluciones.
- La memoria que contiene los cromosomas.

El controlador se encarga de controlar el intercambio de información entre el componente evolutivo y el entorno. A continuación explicamos cómo trabaja el componente evolutivo. La población inicial se genera en el interior de la unidad genética. El componente evolutivo espera hasta que desde el entorno se le solicite un nuevo candidato. Cuando se le solicita, se envía el candidato que se carga en el HW

reconfigurable. Este candidato es evaluado por el entorno. Cuando la evaluación ha terminado, el entorno envía el valor al componente evolutivo y se carga en la memoria de valores de la unidad genética. Este valor es comparado con el mejor valor almacenado. Si el nuevo valor es mejor entonces su cromosoma se almacena en la memoria de cromosomas indicando que es el mejor individuo hasta el momento. Cuando todos los individuos han sido evaluados se inicia el proceso para obtener la siguiente generación. Con este algoritmo se asegura que siempre se conoce el mejor individuo. Este proceso se repite sin fin. Puesto que el componente evolutivo es controlado desde el entorno, la reutilización del componente se obtiene automáticamente.

Como se puede ver, la unidad genética y el controlador son fijos, pero pueden operar bajo diferentes funciones de coste porque esta es implementada en el entorno. Es importante recordar que la unidad genética tiene que diseñarse, es decir sus operadores genéticos tienen que elegirse teniendo en cuenta el tipo de aplicación y el tipo de circuito reconfigurable que se va a usar.

En cuando al controlador, aunque se hable de un ente, es en realidad implementado dos partes, cada una perteneciente a uno de los componentes básicos del sistema. Ambos son similares en cuanto a funcionamiento, con lo que básicamente se diferencian en sus interfaces con el exterior. Con esto se gana en generalidad, pues ninguna de las dos partes depende del controlador externo. En cualquier caso, los detalles de diseño son explicados con más profundidad más adelante.

2.10. APLICACIONES.

Aunque conceptualmente los fundamentos del diseño evolutivo y los sistemas auto-adaptativos on-line son los mismos, lo cierto es que, según sea el objetivo final, las características que los definen son diferentes. En este epígrafe se van a estudiar las diferencias entre estas dos aplicaciones, sus posibles campos de aplicación, sus ventajas, inconvenientes y posibles líneas de estudio.

2.10.1. DISEÑO EVOLUTIVO.

El objetivo de esta aplicación es encontrar automáticamente diseños de circuitos interesantes con comportamientos imposibles de obtener mediante técnicas convencionales. Las principales características del diseño evolutivo son:

- La función de coste es estática, es decir no cambia a lo largo del tiempo.
- La evolución solo se utiliza durante la fase de diseño.
- No importa el tiempo necesitado para evolucionar el circuito, sino su calidad y coste de implementación.
- La evolución es extrínseca y off-line.
- El algoritmo evolutivo no es parte del sistema objetivo.
- El diseñador puede controlar el proceso durante la evolución.

La principal ventaja del diseño evolutivo [8] es que puede explorar un mayor rango de alternativas de diseño, puesto que no está limitado por la rígida estructura del diseño clásico. En éste, el diseñador usa el método de divide y vencerás, cuya principal desventaja es que se basa en unas normas de diseño predeterminadas. Por lo tanto no se estudia ningún diseño que no se ciña a estas normas. Como el diseño evolutivo no usa normas sino sólo comportamientos, se pueden explorar un mayor número de circuitos. Además, el diseño evolutivo no necesita un conocimiento previo para realizar el estudio, solo es necesario fijar mediante una buena función de coste el comportamiento del circuito. En definitiva la principal ventaja del diseño evolutivo es que trabaja pensando en *qué* es lo que va hacer el circuito mientras que las herramientas clásicas piensan en *cómo* implementar el circuito.

Por otro lado, el mayor problema con el que se encuentra el diseño evolutivo es la escalabilidad. Hasta el momento casi todos los trabajos en esta línea han sido realizados con circuitos de muy baja complejidad. Si se quiere aplicar la técnica a circuitos más complejos, hay que salvar dos problemas. Por un lado, la longitud del cromosoma y por otro la complejidad computacional. Como ya se ha indicado con anterioridad, para evolucionar un circuito de unas 100 puertas lógicas se necesitan cromosomas de más de 2000 bits. Hay que recordar que cuanto mayor es el cromosoma, mayor es el espacio de búsqueda y esto repercute tanto en el tamaño de la población como en el tiempo de búsqueda. Referido a esto último, lo normal es que un circuito de aproximadamente 100 puertas lógicas tarde varios días en alcanzar la solución final. Entonces, ¿cuánto tiempo se tardaría en evolucionar un circuito con 10.000 componentes funcionales?

Otro problema que surge es como verificar que el circuito obtenido es correcto. Cuando el circuito tiene muy pocas entradas el método es sencillo pues solo hay que comprobar que cada entrada genera el resultado deseado. Pero, ¿qué hacer en el caso de conjuntos de entradas mucho mayores? La evaluación de cada una de las entradas podría incurrir en un importante coste computacional.

Como resumen se puede afirmar que si la síntesis de circuitos mediante EHW quiere ser una alternativa a las herramientas de diseño clásicas tiene que probar que tiene claras ventajas frente a ellas. Esto no es trabajo fácil, puesto que las herramientas clásicas se encuentran en un estado de desarrollo muy avanzado y han probado a lo largo del tiempo su eficiencia y fiabilidad. Por lo tanto se puede afirmar que el uso de EHW como herramienta de síntesis sólo podría ser útil para desarrollar

circuitos concretos que implicaran una especial dificultad para las herramientas clásicas, o circuitos que las herramientas clásicas per se no pudieran implementar. Quizás el principal campo en el que la síntesis EHW podría aplicarse es en diseño de circuitos analógicos [5].

2.10.2. SISTEMAS AUTOEVOLUTIVOS ON –LINE.

Este tipo de sistemas están pensados para trabajar en entornos dinámicos en los que es necesario alcanzar un elevado rendimiento o adaptabilidad en aplicaciones en las que las especificaciones del problema no son conocidas en la fase de diseño o son cambiadas durante la ejecución. Algunas de sus características son:

- El objetivo final es conseguir una adaptación autónoma y continua en un entorno dinámico. Estas características son típicas de sistemas empotrados.
- La función de coste es dinámica.
- La evolución no tiene fin.
- El tiempo de adaptación es muy importante, puesto que el sistema trabajando on-line tiene que adaptarse en tiempos aceptables para el sistema objetivo.
- La evolución se realiza en HW.
- El algoritmo evolutivo es parte del sistema objetivo, puesto que el sistema tiene que reflejar los cambios en la función de coste.
- El diseñador no puede controlar el algoritmo durante la evolución.

Los sistemas en los que el HW auto-evolutivo on-line sería interesante, son aquellos en los que, por las razones que sean, las condiciones del entorno pueden cambiar de manera descontrolada y por lo tanto no se puede saber de antemano cual es la configuración que el HW podría necesitar y no existe la posibilidad de un reajuste o rediseño del mismo. Un ejemplo podrían ser las misiones espaciales en las que la nave puede pasar por zonas en las que se producen rápidas variaciones de temperatura o por zonas de altas radiaciones o, en misiones de larga duración, en las que se puede producir un efecto de envejecimiento de los circuitos. En todos los casos el comportamiento del circuito varía de manera impredecible. El EHW podría solucionar el problema adaptándose a los cambios del entorno.

También sería muy interesante su utilización para generar sistemas tolerantes a fallos. Existen una clase de sistemas en los que reparar o reemplazar el HW no es posible. Generalmente estos sistemas tienen unas características especiales, como por ejemplo tienen poco espacio y restricciones de peso por lo que solucionar los fallos mediante redundancia no es posible.

Los diseñadores saben que los sistemas tarde o temprano fallan. Algunas veces los fallos son pequeños y aparentemente no afectan al funcionamiento del sistema. Otras veces los fallos son lo suficientemente graves como para afectar al sistema en su totalidad. Esta es la razón por la que los diseñadores intentan implementar sistemas tolerantes a fallos, es decir sistemas que pueden seguir trabajando con un fallo, aunque quizás perdiendo algo del rendimiento. Muchos de los sistemas tolerantes a fallos ocultan éste, pero no lo recuperan, por lo que el problema persiste y puede acabar dañado al resto del sistema. Evidentemente lo ideal es detectar el fallo y reparar el sistema tan rápido como sea posible. Este proceso tiene dos partes, una primera de detección y aislamiento del fallo, y una segunda de recuperación del fallo. Desde el punto de vista del EHW es interesante la recuperación de fallo.

El método más habitual de recuperación de fallos es la redundancia de HW, que consiste en tener HW replicado y cuando se produce un fallo sustituir ese circuito por otro idéntico y seguir trabajando hasta que se pueda sustituir el HW dañado. El problema es que este método no se puede usar siempre, por ejemplo en sistemas que tienen el espacio limitado porque lo necesitan para otros módulos y no pueden añadir HW replicado o en sistemas en los que pueden aparecer cambios no previstos en el entorno como entornos de alta radiación que puede afectar a los dispositivos MOS o altas temperaturas que pueden afectar a los conversores de voltajes. En estos casos no es eficiente cambiar el HW por otro puesto que las condiciones desfavorables del entorno pueden persistir.

Los sistemas auto-evolutivos on-line suelen ser sistemas de tiempo real, es decir el fallo o la variación del entorno tiene que localizarse, aislarse y recuperarse en tiempos predefinidos. No es importante que esto se haga en el menor tiempo posible pero sí que se haga en el tiempo prefijado. Por lo tanto, aunque el EHW parece una buena aproximación, el diseñador siempre debe tener en cuenta cual es el uso final del sistema y asegurarse que su propuesta es capaz de evolucionar en el tiempo previsto. Esto es importante porque la evolución se suele parar o bien porque se alcanza el comportamiento deseado o bien porque se ha consumido un determinado número de generaciones. En el primer caso se debe asegurar que el circuito deseado se consigue dentro del tiempo predefinido. En el segundo caso pudiera ser que el circuito obtenido no solucionara el problema.

Evidentemente en este tipo de sistemas es mejor usar evolución intrínseca que evolución extrínseca por dos razones. Como hemos visto, estos sistemas no suelen tener mucho espacio para HW, y por lo tanto es poco probable que tenga un microprocesador de propósito general como el Pentium o una estación de trabajo Linux para realizar la evolución. Todo lo contrario, el sistema probablemente usará un microprocesador low-end en lugar de usar uno de alto rendimiento. Este tipo de procesadores son más lentos y pueden direccionar menos memoria, por lo tanto la evolución extrínseca sería más lenta y menos precisa que la evolución intrínseca. Sin embargo, estos procesadores son capaces de trabajar correctamente con cualquier tipo de algoritmo evolutivo. La segunda razón es el problema de la portabilidad, es decir, la solución obtenida por simulación HW podría no funcionar cuando se implementara en el HW. Una vez más hay que recordar que en el método intrínseco

hay que reconfigurar el HW para cada una de las evaluaciones, y que la reconfiguración de HW requiere su tiempo, pudiendo impedir que se alcanzara el circuito correcto en el tiempo especificado.

Parece que la principal aplicación del EHW podría ser en circuitos adaptativos donde el circuito modifica su comportamiento autónomamente, sin intervención exterior, mientras que opera en entornos reales. Esta adaptación podría ser bien para generar una nueva función, bien para recuperarse de un fallo, o bien para mantener la funcionalidad cuando se produce un cambio desconocido en el entorno. El cambio de funcionalidad no parece tener mucho interés si este nuevo comportamiento se conoce de antemano, por lo tanto las dos últimas aproximaciones parecen las más interesantes aunque todavía hay muchos problemas sin resolver. A continuación se van a hilvanar algunos de ellos [5]:

- Generalmente el EHW no trabaja aislado, sino en un sistema con otros dispositivos, configurables o no. Esto quiere decir que los cambios que se produzcan en el EHW pueden afectar al sistema. Una pregunta que habría que responder es ¿qué se debe hacer con el resto del sistema mientras que el EHW está evolucionando para adaptar su comportamiento? Indudablemente parece que en el peor de los casos se debería aislar el resto del sistema para evitar que se dañara.

- Hay que tener claro que no siempre el circuito final obtenido tendrá el mejor comportamiento posible y por lo tanto también puede dañar el comportamiento total del sistema.

- Hay que tener en cuenta que en los sistemas on-line, a diferencia de lo que ocurre en los sistemas off-line, el tiempo de evolución es muy importante. El problema aparece porque la evolución se basa exclusivamente en la población y puede ser demasiado lenta para estos sistemas. Por lo tanto, parece que los sistemas más apropiados serían sistemas mixtos de búsqueda global-local, es decir que utilizan el conocimiento local para acelerar esta evolución [8].

- Como ya se ha indicado con anterioridad, los circuitos complejos no pueden usar una especificación detallada del comportamiento porque su evaluación tardaría demasiado tiempo. En estos casos se suelen usar conjuntos de entrenamiento para realizar la evolución, pero, ¿podemos asegurar que el circuito final, con el mejor valor de la función de coste, es la mejor generalización del circuito? Es decir, frente a otro conjunto de entradas, ¿el circuito se comportaría igual de bien?

- Los diseñadores deberían tener en cuenta que el daño producido puede ser tan importante que no se pueda reestablecer completamente la funcionalidad del sistema, en estos casos el EHW debería ser capaz de buscar una configuración capaz de recuperar, como poco, parte del rendimiento del sistema.

3. METODOLOGÍA Y DESARROLLO

A lo largo de este apartado se pretende describir manera más precisa cómo se ha desarrollado el sistema, desde los principios en los que se basa su planteamiento hasta su estructura modular. Como a lo largo de todo desarrollo, existen una serie de compromisos de diseño, cuyas ventajas e inconvenientes hay que sopesar y posteriormente elegir la mejor opción. Éstos son introducidos y comentados. Así mismo, y como no podía ser de otra manera, se explica también el funcionamiento de los dispositivos sin los cuales sería imposible llevar a cabo el hardware evolutivo, las FPGA. Las introducimos de forma genérica, y profundizamos más adelante en la FPGA y la placa de desarrollo con la que contamos, así como las herramientas en las que nos hemos apoyado.

La funcionalidad del sistema es la de un reconocedor de patrones usual de tres bits, de forma que sea un objetivo sencillo pero interesante desde el punto de vista de un estudio o investigación. Suponiendo una entrada serie y un patrón de tres bits definido mediante los botones de la placa, el sistema será capaz de desarrollar una solución eligiendo entre las celdas funcionales y la interconexión programable del módulo de procesamiento. Interactuando con el sistema podremos recomenzar la

búsqueda, descartar el mejor circuito o mostrar el mejor evolucionado hasta el momento.

Grosso modo, podemos definir que la característica principal de este diseño no es su diseño novedoso, ni óptimo en tiempo o coste, sino que es que se trata de un sistema que automáticamente busca la solución al problema planteado. Además, se trata de HW evolutivo on-line puesto que la evolución se realiza dinámicamente sobre los mismos componentes que después implementarán el circuito final. La evolución en sí es también auto-adaptativa, puesto que en caso de cambiar las entradas externas de las que depende, el mismo se redefinirá para buscar la solución al nuevo problema. Además, es posible que durante la evolución o la ejecución normal se produzcan fallos en los diferentes componentes del circuito objetivo. El propio sistema dará cuenta de ello y propondrá una posible solución en la que esas celdas son ignoradas.

Entonces, simplemente definiendo el cómo, es decir, las entradas del sistema y cómo deben comportarse sus salidas, se evolucionan posibles circuitos hasta dar con el correcto. Éste será tal que para todas las entradas definidas, produzca las salidas pertinentes. Los circuitos se van proponiendo hasta encontrar uno apto, o hasta concluir la fase de evolución por haber sobrepasado el límite de generaciones en la evolución. Las diferentes posibles soluciones quedan definidas en forma de cromosomas, que son caracterizaciones inequívocas de circuitos digitales. Los cromosomas como ya hemos visto van a ser son cadenas binarias, en los que cada bit representa parte del enrutamiento o de las funciones que implementa el sistema. Siguiendo el esquema de los Componente Evolutivos de [14], [15] y [16], el núcleo del sistema esta compuesto por una parte de proceso y otra de enrutado reconfigurables. Es por eso que se trata de un nivel hardware reconfigurable sobre el propio de la FPGA. Es decir, que puede verse como una arquitectura reconfigurable impuesta sobre la de nuestra FPGA, pero sobre la que se tiene un control total y la posibilidad de reconfigurar dinámicamente y de forma rápida.

Este enfoque se basa en los Virtual Reconfigurable Circuits definidos en [14], [15] y [16], y como aproximación resulta muy interesante dentro del campo del hardware reconfigurable por su versatilidad y sencillez. El hecho de tener el control y conocimiento totales de su funcionamiento son los elementos clave de su éxito. Además de permitirnos una rápida reconfiguración, permite definir bloques funcionales específicos para nuestra aplicación. Y es que la evolución va a ser llevada a cabo a nivel funcional, es decir, no se trata de definir transistores ni puertas lógicas, sino de describir las funciones que los bloques funcionales podrán llevar a cabo. En cualquier caso, estos detalles son tratados de formas más profunda en los siguientes apartados.

3.1. INTRODUCCIÓN A LAS FPGA. XILINX VIRTEX II PRO Y PLACA DE DESARROLLO XUPV2P.

Las FPGA son dispositivos semiconductores que contienen bloques de lógica cuya conexión y funcionalidad pueden ser programadas tantas veces como el usuario/desarrollador quiera. Su ámbito de aplicación es el mismo que el de los ASIC (Application Specific Integrated Circuit), pero sus características son diferentes, pues las FPGA son más lentas, consumen más y tienen limitaciones de espacio. Sin embargo, sus ventajas son una gran versatilidad y su bajo precio por unidad y al desarrollar sistemas, por lo que hoy en día son ampliamente utilizadas.

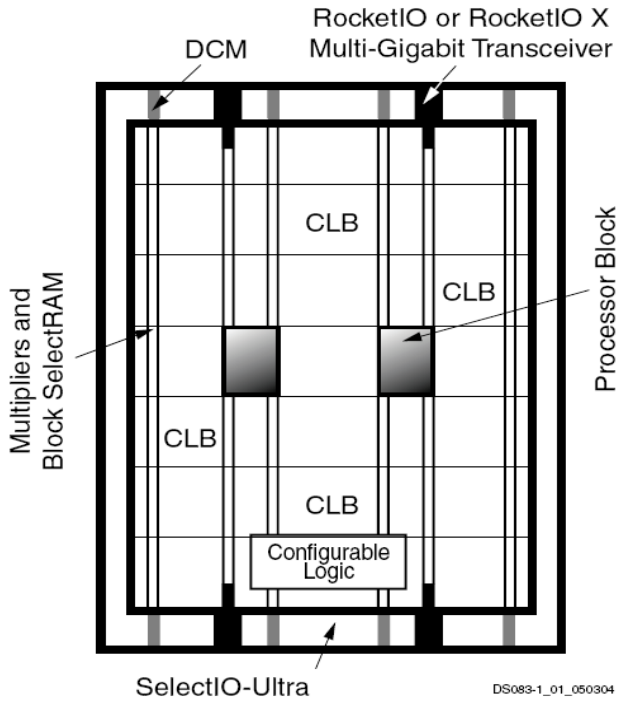

Ilustración 3. Arquitectura modular de la FPGA Virtex II Pro de Xilinx (Fuente: Xilinx).

En la Ilustración 3 podemos ver un esquema de la arquitectura de la FPGA con la que contamos en nuestro trabajo de investigación, la Virtex-II Pro de Xilinx [65], que hoy en día es el principal fabricante de FPGA a nivel mundial. Como podemos apreciar, se compone principalmente de:

- CLB. Configurable Logic Blocks. Son los bloques que llevan a cabo la carga funcional del sistema. Constan en nuestro caso de 4 slices cada uno, y son, por supuesto, programables en casi todos sus parámetros. Cada uno de los slices, como se observa en la Ilustración 4, esta compuesto por dos módulos funcionales básicos implementables en forma de LUT (LookUp

Table) de 4 entradas, en forma de registro de desplazamiento de 16 bits o de memoria SelectRAM distribuída también de 16 bits. Los dos elementos de memoria que se ven en el margen derecho son implementados según la decisión del usuario, pudiendo ser utilizados como latches activados por nivel o flip-flops por flanco, ambos de diversa índole. Además, cada slice cuenta con una gran cantidad de lógica de control y acarreo para llevar a cabo funciones más complejas o con mayor número de entradas.

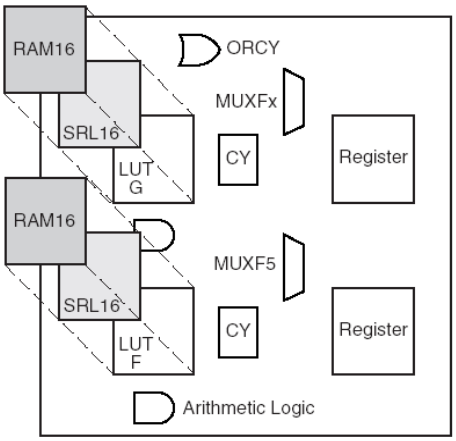

Ilustración 4. Arquitectura interna de cada uno de los slices que componen un CLB en la Virtex II Pro (Fuente: Xilinx).

- IOB. In/Out Blocas. Esta FPGA cuenta con varios tipos de módulos de E/S. Aunque cambian sus especificaciones, su función es la de proporcionar una interfaz entre el exterior del dispositivo y la red de interconexión interna.

- DCM. Digital Clock Managers. Módulo que gestiona las señales de reloj, de modo que lleguen a la vez a todos los puntos del dispositivo y que a su vez puede generar otras a partir de las recibidas.

- POWERPC 405E. Power PC Hard Core Processor 405E de IBM. Cuenta con dos procesadores empotrados dentro del dispositivo, por lo que resulta apto para el desarrollo de sistemas empotrados basados en co-diseños HW-SW. Son procesadores RISC (Reduced Instruction Set Architecture) de 32 bits que alcanzan una frecuencia de trabajo máxima de 400MHz. Su diseño se basa en la arquitectura Power de IBM, con una segmentación en 5 etapas y con una memoria caché de primer nivel de 16KB separada para instrucciones y datos.

- BRAM. Block Random Access Memory. Incorpora a su vez bloques de 18K de RAM distribuida por columnas entre el resto de elementos. La FPGA con código XCV2VP30, dispositivo con el que contamos, implementa 136 bloques de memoria RAM dedicada de 18Kb, es decir, 2448KB distribuidos

por todo el dispositivo. En caso de necesitar más memoria que la ofertada por uno de esos componentes, es fácilmente ampliable si se comunican varios módulos en cascada.

- Multiplicadores de 18 bits. se trata de bloques dedicados capaces de multiplicar dos números de 18 bits con signo de forma rápida y con bajo consumo si se compara con un multiplicador similar sintetizado en slices. Pueden usarse independientemente o junto a bloques de SelectRAM con los que se sincronizan perfectamente.

- Red de Interconexión. los recursos locales y globales de enrutado de la Virtex-II están optimizados para conseguir la máxima velocidad de transferencia posible sin comprometer para ello la flexibilidad. Es por ello que la arquitectura de la red de interconexión local se distribuye junto al resto de los elementos de la placa, a través de switches idénticos, como se puede apreciar en la Ilustración 5. Globalmente, estos switches quedan unidos mediante los canales globales de enrutado desplegados en la placa. Dependiendo de las necesidades de conexión entre dos elementos dados, se elegirá el mejor camino entre la líneas existentes, ya sean horizontales o verticales, largas, hex, directas o dobles y con mejores o peores prestaciones.

La FPGA basa su funcionamiento en la cadena de configuración o bitstream, cargada por el usuario. Ésta es almacenada en la memoria SRAM que incorpora para tal fin. Esta cadena binaria de configuración es la que denota las funciones, enrutamiento y demás información necesaria.

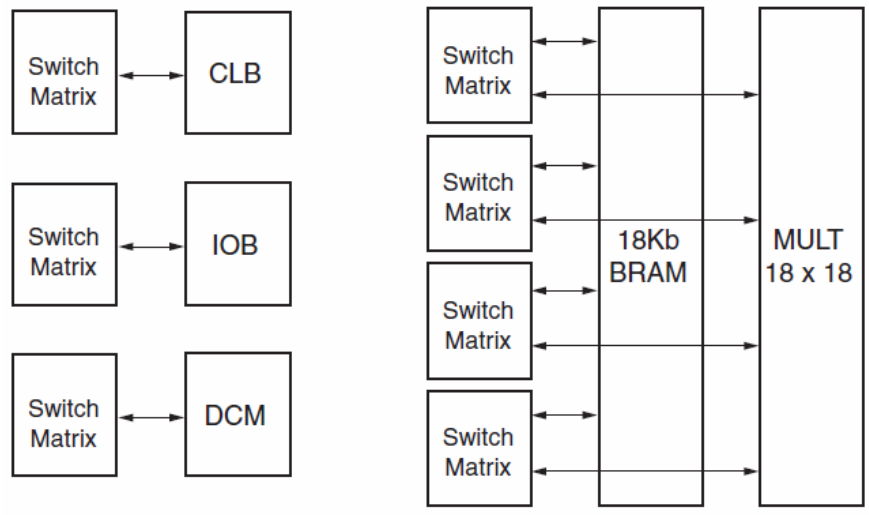

Ilustración 5. Tecnología activa de interconexión (Fuente: Xilinx) .

Una vez presentada la FPGA con la que vamos a trabajar, queda introducir la placa de desarrollo que va a servir de plataforma a la investigación. Contamos con

dispositivos basados en Virtex II Pro correspondientes al programa universitario de Xilinx, que se comercializan en placas de desarrollo con nomenclatura XUPV2P, acrónimo con el mismo significado. Según [65], podemos especificar que cuentan principalmente con:

- FPGA Virtex-II Pro y dos Power PC 405E empotrados.
- Hasta 2GB de memoria DDR SDRAM.
- Dispositivo de red Ethernet 10/100 integrado.
- Puerto serie RS-232.
- Dos puertos serie PS-2.
- 4 LED conectados a la FPGA.
- 4 botones conectados a la FPGA.
- 5 pulsadores conectados a la FPGA.
- CODEC AC-97 para audio, con amplificador incorporado y salida audio.
- Entrada de sonido para, por ejemplo, un micrófono.
- Salida XSGA.
- Tres puertos Serial ATA.
- Reloj del sistema de 100MHz.

En la Ilustración 6 podemos apreciar con más detalle cómo esta dividido el dispositivo. Se puede observar que cuenta con la FPGA como elemento principal, una serie de entradas entre las que destacan las diferentes tensiones a las que es capaz de trabajar, la frecuencia de trabajo y la procedencia del bitstream de configuración, así como los módulos más arriba enumerados. El hecho de contar con esa gran cantidad de interfaces, tanto simples como complejos la hace versátil y a la vez potente, siendo de gran utilidad como plataforma de aprendizaje, pero también para desarrollos bastante más extensos y complicados. El aspecto físico de la placa podemos apreciarlo en la Ilustración 7.

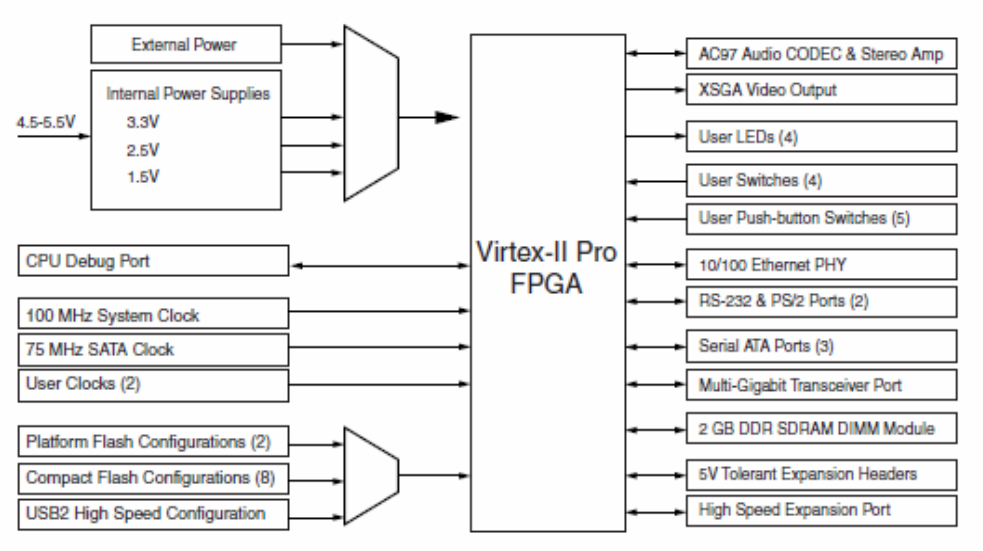

Ilustración 6. Diagrama modular de los principales bloques de la placa de desarrollo XUP Virtex-II Pro (Fuente: Xilinx).

Mención especial requiere el Microblaze, un soft-core desarrollado por Xilinx [63], que aunque no se trate como un componente físico aislado al ser sintetizado en las propias celdas de la FPGA, su importancia hace que le dediquemos unas líneas. Se trata de un procesador RISC (Reduced Instruction Set Computer) que cuenta con 87 instrucciones diferentes. Lo encontramos disponible en las placas de las familias Spartan y Virtex, y su arquitectura interna es de tipo Harvard, es decir, cuenta con interfaces de bus y espacios de memoria disjuntos de datos e instrucciones. Para llevar a cabo el co-diseño de nuestro proyecto, vamos a hacer uso de este procesador.

La configuración del dispositivo reconfigurable se puede llevar a cabo siguiendo varios enfoques [65], seleccionables desde la propia placa a través de uno de los switches, que, coloreados en rojo se encuentran en la parte derecha de la placa según observamos en la Ilustración 7.

- JTAG (Joint Test Action Group). El proceso de configuración sigue las pautas explicadas en este estándar de validación de módulos y PCB (Printed Circuit Board). La fuente de datos de la configuración puede provenir de la tarjeta de datos Compact Flash conectable (Ilustración 7, parte derecha) a la propia placa o desde una fuente JTAG externa, como puede ser un cable PC4 (Paralell Conection 4) o el interfaz USB (Universal Serial Bus). Proporciona una velocidad de carga de datos baja, aunque por otra parte es el modo por defecto y también el más extendido.

- selectMAP maestro. Es el estándar definido por Xilinx para la carga de bitstreams de configuración desde su herramienta propia iMPACT. La fuente de los datos es la PROM (Programmable Read-Only Memory)

integrada en la placa. Ésta proporciona gran velocidad de transferencia y puede ser configurada por el usuario.

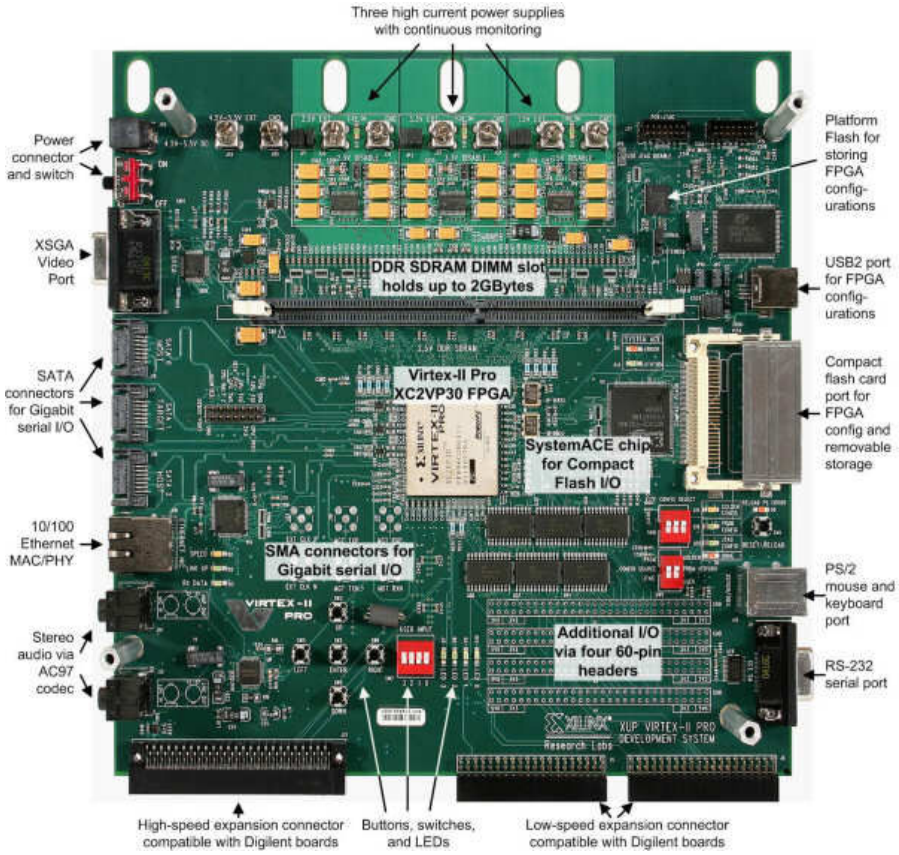

Ilustración 7. Placa de desarrollo XUPV2P perteneciente al Xilinx University Program.

Ya que el Hardware Evolutivo es un campo de investigación que al parecer, hoy en día, se encuentra puramente en entornos de investigación universitarios, es lógico que esta placa ya haya sido igualmente la plataforma de partida en otras ocasiones para investigaciones similares [17], [31].

3.2. HERRAMIENTAS DE DESARROLLO.

Es de agradecer en este tipo de trabajos la inclusión de un pequeño apartado que aborde las herramientas de desarrollo utilizadas en la consecución del proyecto.

Creemos que puede resultar muy útil al lector que no este muy habituado con estos entornos y, quizás servir de aclaración para el potencial diseñador que continúe el camino empezado aquí empezado, dentro de esta interesante línea de investigación. Las enumero a continuación:

- Xilinx ISE Foundation 10.1. El entorno Integrated Software Environment [61], [62] en su versión 10.1 ha sido la aplicación elegida para el desarrollo y síntesis sobre FPGA de los módulos hardware que contiene el proyecto. Esta herramienta es un entorno que integra todas las utilidades necesarias para diseñar sistemas sobres dispositivos lógicos reconfigurables, los PLD (Programmable Logic Device). Es el mayor exponente de su clase en el campo del diseño digital de circuitos gracias a su potencia y facilidad de uso, y por pertenecer al líder del mercado en la actualidad, Xilinx Inc. Recibiendo código HDL y algunos ficheros de restricciones y configuración, lleva a cabo la síntesis del sistema sobre el CPLD elegido, pudiendo decidir la asignación de pines de E/S, la generación de informes de coste y consumo de potencia e incluso la simulación temporal, ésta gracias al simulador que incluye.

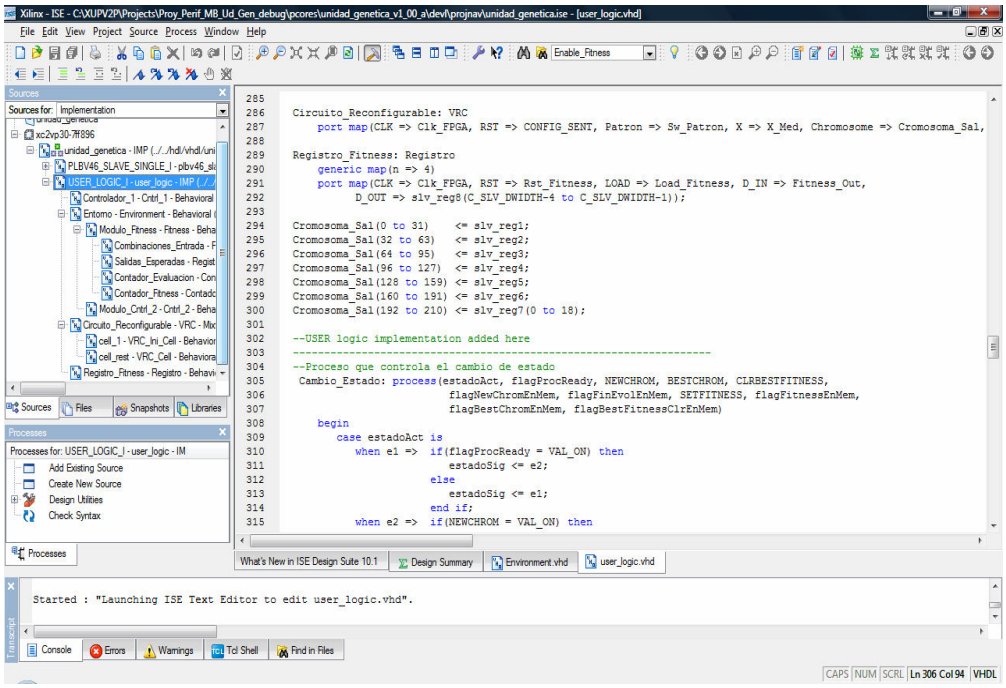

Ilustración 8. Captura de pantalla en la que se muestra el desarrollo del proyecto aquí expuesto, utilizando Xilinx ISE 10.1.

- Xilinx EDK 10.1. La herramienta de diseño de sistemas empotrados Embedded Development Kit es, según puede comprobar el lector en nuestra bibliografía, una suite muy utilizada en el desarrollo de hardware evolutivo [31]. Esta compuesta principalmente por [64]:
    - XPS. Núcleo de la herramienta, se describe a continuación.
    - Entorno de desarrollo de software SDK. El Software Development Kit se basa en el IDE (Integrated Development Environment) de Eclipse. Incluye el compilador de C/C++ de GNU gcc así como el depurador del sistema operativo propio, XMD (Xilinx Microprocessor Debugger).
    - Documentación y código de IP (Intellectual Property) necesarios para el desarrollo de sistemas empotrados basados en PowerPC y/o Microblaze, sobre las FPGA propias.
    - Núcleos y utilidades para desarrollos basados en sistemas operativos sencillos sobre los procesadores ya nombrados.

Para más información sobre la herramienta, se remite al lector al manual de utilización y guía de diseño de sistemas creada por el propio fabricante, Xilinx [64].

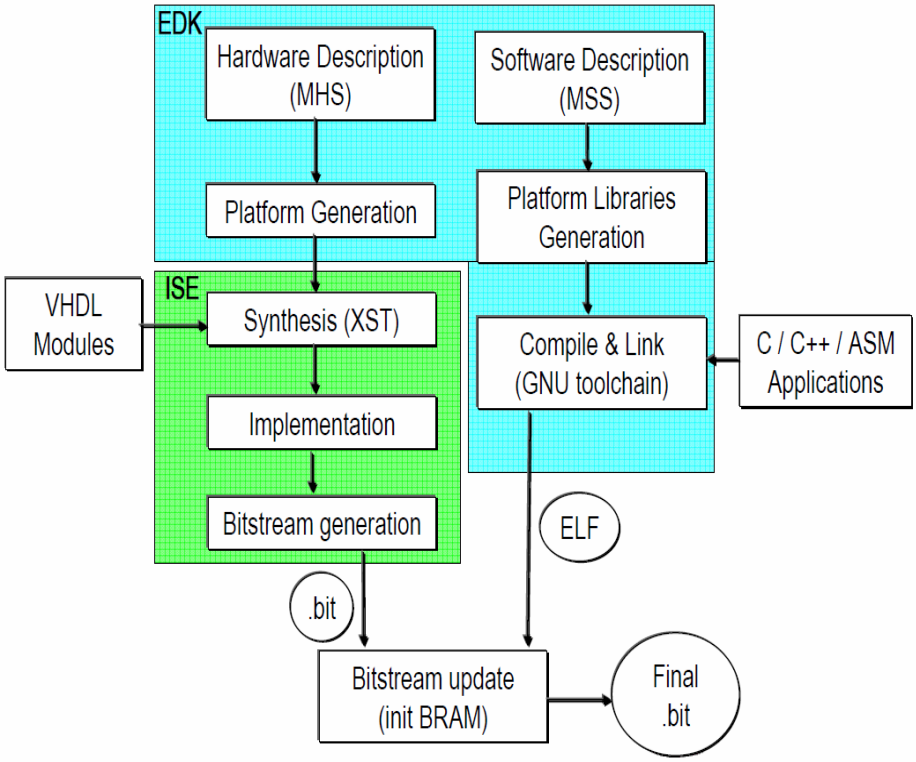

Ilustración 9. Diagrama de flujo de los procesos y herramientas que se utilizan en la generación del archivo de configuración de una FPGA (Fuente: Curso DISOC'08, UCM).

Para diferenciar el límite entre el campo de actuación del EDK y del ISE, se incluye la Ilustración 9. Ésta muestra de forma modular los procesos que se ejecutan durante la formación del bitstream de configuración que se carga en la FPGA objetivo. Éste puede estar formado sólo por descripciones hardware cuyo origen sea un fichero MHS (Microprocessor Hardware Specification) o módulos propios, o por éstos anteriores más una serie de aplicaciones software traducidas y especificadas en el fichero MSS (Microprosessor Software Specification). Según sea el caso, los procesos que se ejecutan serán dirigidos por el ISE (verde), o por éste y el EDK (azul). Continuando con el listado de utilidades y tecnologías:

- Xilinx Platform Studio. Permite diseñar el sistema empotrado al completo, pues contiene un catálogo de drivers, bibliotecas y compiladores para los procesadores soportados. Facilita en gran manera la creación de proyectos y la inclusión de ciertos componentes basándose en asistentes muy intuitivos. En la Ilustración 10, se puede apreciar una captura de pantalla del entorno de desarrollo XPS. En él se visualizan los diferentes módulos incluidos en el proyecto y sus interfaces de comunicación con los buses de datos (parte central y derecha), las aplicaciones SW que se pueden ejecutar sobre el procesador integrado (parte izquierda) y una consola (parte inferior) para seguir la traza de los procesos que se realizan.

- Xilinx iMPACT. Es una utilidad que permite configurar y programar ágilmente los PLD de Xilinx, o memorias PROM (Programmable Read Only Memory) que serán utilizadas como base para configuraciones posteriores. Puede ser ejecutado desde su propia interfaz gráfica o por lotes desde otras aplicaciones, como ISE o EDK. A través de un cable serie, una conexión USB A-B o incluso una memoria flash se pueden programar las FPGA siguiendo uno de los siguientes esquemas de configuración:

    - Xilinx Slave Serial. modo de transmisión serie sencillo, de un solo bit y con una tasa de transferencia de hasta 66 Mbits.

    - JTAG o Boundary Scan. acrónimo de Joint Test Action Group, es también un estándar de IEEE, concretamente el 1149.1. Es utilizado para validación y depuración de circuitos integrados y módulos hardware. Para ello, incluyen diversos registros de estado junto a componentes y pines de E/S para el control y la validación de valores.

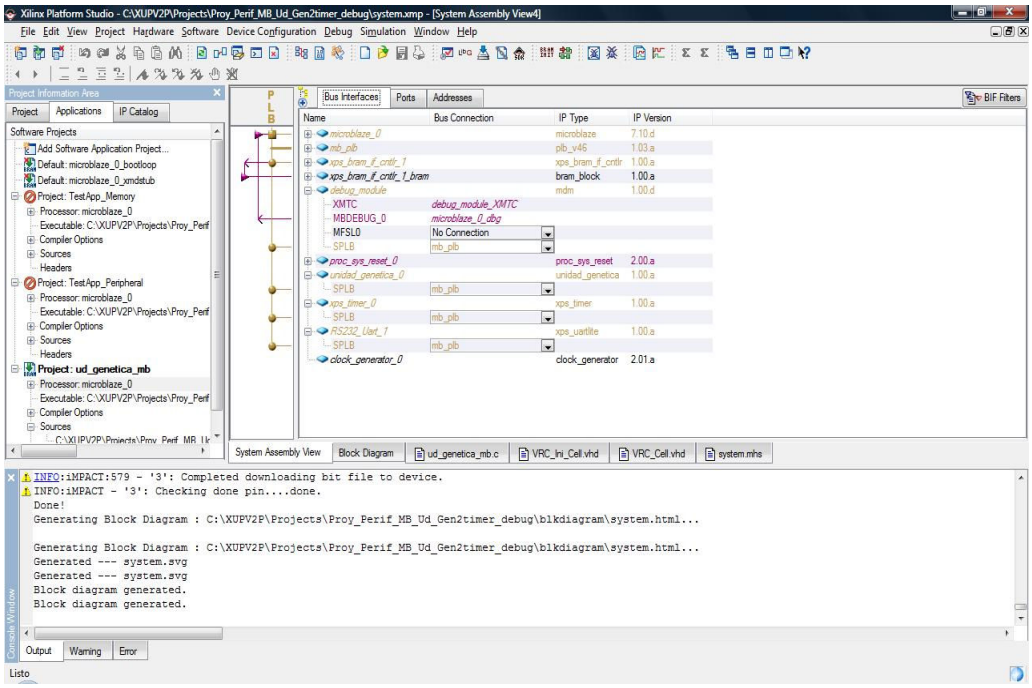

Ilustración 10. GUI (Graphical User Interface) de la herramienta EDK de Xilinx.

- Mentor Graphics ModelSim SE Plus 6.5. se trata de una plataforma de simulación y depuración electrónica muy potente e intuitiva enfocada a ASIC (Application-Specific Integrated Circuit) y FPGA. Permite, desde la simulación de pequeños módulos hardware como podría ser un contador ascendente/descendente hasta un diseño de un sistema empotrado con un microprocesador MicorBlaze/PowerPc y varias memorias con diferentes tiempos de acceso. En cuanto a los lenguajes de entrada, soporta los más importantes, como son VHDL, Verilog, SystemVerilog y SystemC.

En la Ilustración 11 podemos apreciar con más detalle el resultado de una de las simulaciones llevadas a cabo con la herramienta ModelSim. Se trata de la simulación temporal del banco de pruebas implementado para la depuración del módulo propietario Fitness. Como vemos, a la izquierda se listan las señales que en la parte derecha son simuladas. De esta manera, es posible validar y depurar cada uno de los módulos hardware por separado, e ir aumentando la abstracción siguiendo así un esquema de desarrollo bottom-up. De esta manera se consiguen disminuir los errores al máximo cuando va a depurarse el sistema en su conjunto.

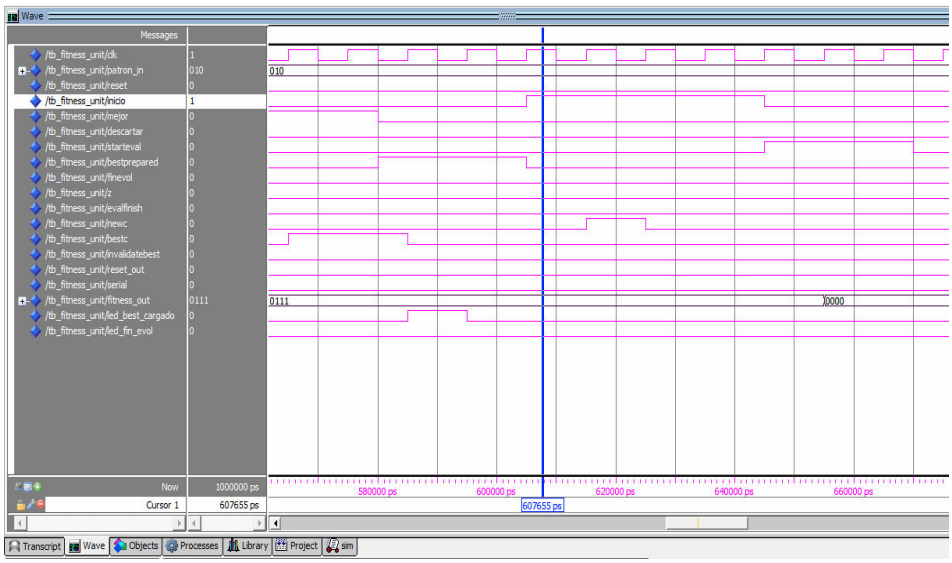

Ilustración 11. Detalle del diagrama temporal del módulo de Fitness de nuestro sistema.

- Xilinx Microprocessor Debugger (XMD) y GNU Debugger (GDB). Con estas dos herramientas es posible depurar los errores de código SW que se cometen durante el desarrollo. GDB es el depurador de C por antonomasia, se distribuye con licencia libre GNU y es ampliamente utilizado, por lo que ya estábamos acostumbrados. Esta herramienta utiliza la aplicación XMD como motor de comunicación hacia los diferentes microprocesadores destino. Provee un interfaz remoto de funcionamiento basado en sockets y en protocolo TCP (Transmission Control Protocol). Es por ello que para poder lanzar la aplicación de depuración GDB es necesario haber ejecutado antes la herramienta XMD, sobre la que actúa.

- HyperTerminal. Aplicación suministrada con el sistema operativo Windows que permite conectar y monitorizar las comunicaciones entre dos dispositivos utilizando para ello el puerto serie del ordenador. Permite así conectarnos a otros equipos y controlar e imprimir la información recibida por dicho puerto proveniente por ejemplo de un modem, un robot o una FPGA. Las herramienta EDK viene provista de un IP de comunicación serie UART (Universal Asynchronous Receiver-Transmitter), que será necesario instanciar para tener acceso a la comunicación. Además, la configuración en ambas partes deberá ser configurada de manera similar, es decir, tanto en la unidad UART dentro de nuestro hardware reconfigurable, como en el Hyperterminal de nuestro PC. En nuestro caso hemos elegido trabajar a 9600 baudios de velocidad, sin bit de paridad y con 8 bits de datos.

En la Ilustración 12 podemos observar la información recibida en el HyperTerminal y proveniente del módulo de comunicación UART del sistema empotrado. Se trata de una traza de la ejecución que se lleva a cabo en el procesador, básicamente consta de una parte que muestra la evolución de los

diferentes cromosomas y de otra que ofrece el seguimiento de la comunicación entre éste y el módulo IP (Intellectual Property) desarrollado. Como esta comunicación se lleva a cabo mediante el intercambio de flags a través de mapeado en memoria, se muestran esas posiciones para el seguimiento y depuración.

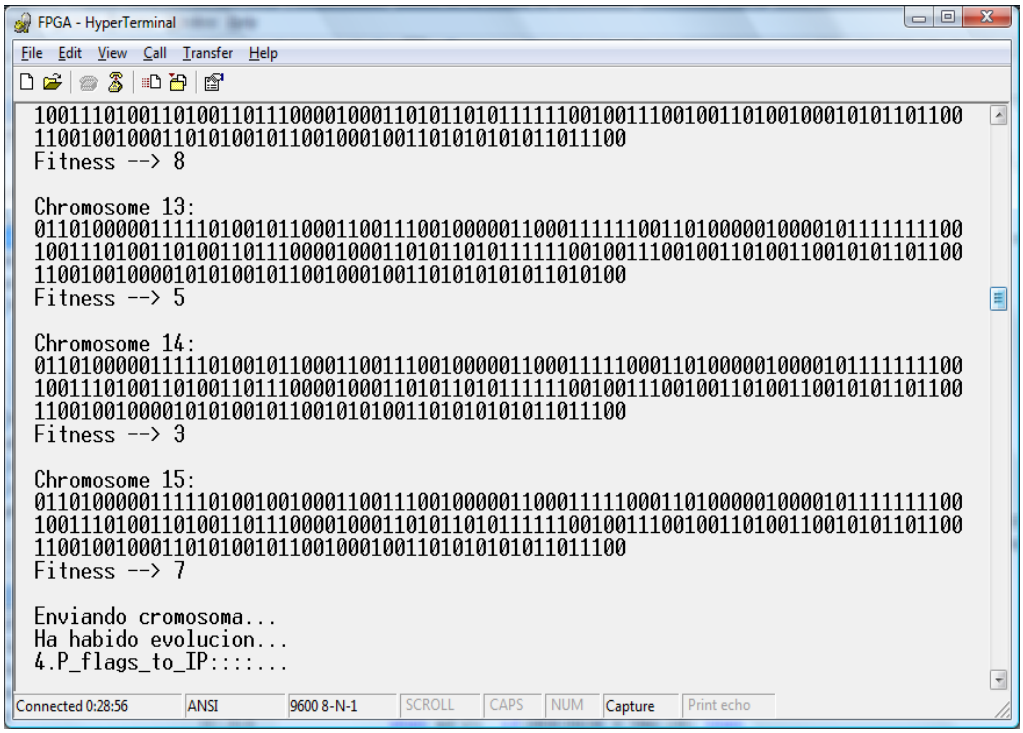

Ilustración 12. Captura de pantalla de la información recibida por el puerto serie, desde la FPGA hasta nuestro PC. Se visualiza toda la información relativa a evolución de cromosomas y comunicación entre módulos y procesador de cara a depurar con todas las garantías.

- Lenguaje C. Se trata de uno de los lenguajes de programación más potentes y extendidos hoy en día. Esta débilmente tipificado y es considerado como de medio nivel, un estrato intermedio entre, por ejemplo, el lenguaje ensamblador (bajo nivel) y el lenguaje Java (alto nivel). En cualquier caso, tiene algunas características de bajo nivel, como el direccionamiento de memoria. Paralelamente, algunas ampliaciones o derivaciones permiten la programación a alto nivel, como ocurre con el C++.

El C ha sido el lenguaje elegido por Xilinx para desarrollar software sobre los procesadores empotrados en sus desarrollos. Es por ello que la herramienta EDK esta provista de un compilador perteneciente al archiconocido GCC (GNU Compiler Collection), de libre distribución.

Las bibliotecas de programación C accesibles por el programador para los desarrollos con procesadores empotrados, PowerPc o Microblaze, se describen en [67]. Se trata de bibliotecas estándar, así como funciones más específicas para acceder a algunos periféricos. Son provistas por la herramienta EDK y configuradas

por una pequeña utilidad llamada Libgen. El fichero que se ejecutará en el sistema en desarrollo será el resultado de la fase de conexión o Link. Ésta recibe, precisamente, información de las librerías a través del archivo proveniente de Libgen, y de la salida de la compilación. Podemos ver el proceso esquematizado en la Ilustración 13.

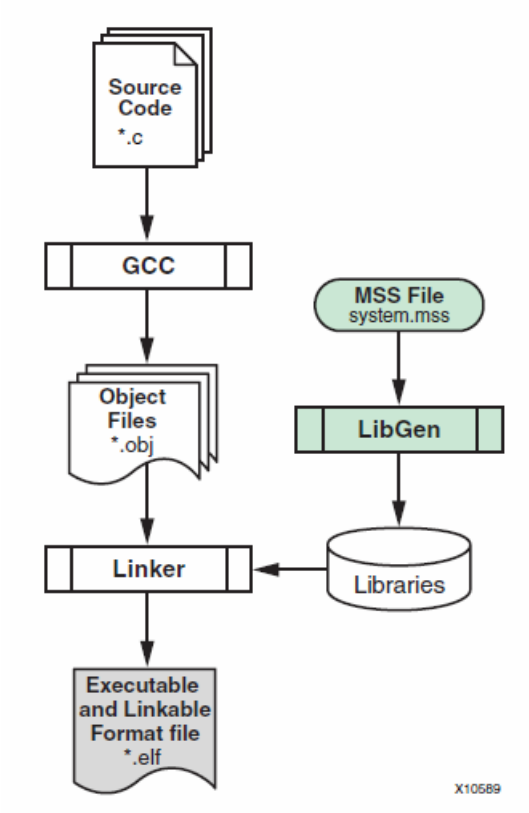

Ilustración 13. Elementos y fases en la generación del fichero ejecutable en el procesador (Fuente: Xilinx).

- Lenguaje VHDL. Acrónimo de VHSIC (Very High Speed Integrated Circuit) y Hardware Description Language, pertenece a un tipo de lenguajes de programación llamados "de descripción hardware" (HDL). Éstos tienen similitudes con los lenguajes de programación de alto nivel, permitiendo un nivel de abstracción elevado cuando se pretende configurar el hardware sin la necesidad de entrar a diseñar a nivel de puerta lógica. Permiten, sin necesidad de utilizar esquemáticos, la documentación de las conexiones y comportamientos de un circuito electrónico. Son muy útiles por la abstracción en la que se basan y porque son independientes del hardware final y a la vez muy modulares. Ejemplos de estos lenguajes son VHDL, Verilog o ADA, siendo los dos primeros los más utilizados en proyectos como el que nos atañe. Permiten, además, llevar a cabo simulaciones lógicas y temporales, pues hay instrucciones para inducir retardos en sus elementos.

El VHDL esta definido como estándar del IEEE (Institute of Electrical and Electronics Engineers) desde 1987, datando su última revisión del año 2008 aunque todavía no ha sido aprobada [68]. A la hora de diseñar un circuito, se puede hacer desde varios enfoques:

- Funcional. Describe cómo se comporta el circuito. Se basa en una serie de instrucciones de ejecución concurrente, y en los procesos. Se parece bastante a los lenguajes de programación imperativa como puede ser el C.
- Flujo de datos. Describe asignaciones concurrentes de señales.
- Estructural. Un circuito es descrito a partir de la instanciación de otros componentes ya desarrollados o pertenecientes a alguna biblioteca predefinida. Esta manera de diseñar es, entonces, jerárquica.
- Mixta. Parte de una combinación de las anteriores.

En nuestro caso, el enfoque seguido a la hora de implementar el sistema ha sido mixto. El proceso de desarrollo ha sido bottom-up, es decir, se ha empezado codificando y validando los módulos más específicos y que realizan funciones más simples y se ha ido subiendo en la abstracción hasta llegar a los más superiores. Es por ello que las instanciaciones superiores requieren un diseño estructural, claro que la funcionalidad de cada módulo en concreto ha sido descrita de manera funcional, basándonos en procesos.

3.3. CO-DISEÑO HW-SW.

Los problemas en computación se resuelven mediante algoritmos, que vienen a ser listas bien definidas, ordenadas y finitas de instrucciones. Pasando por una serie de estados que llevan a cabo esas operaciones, se obtendrá un resultado como solución. Conocida la estructura del algoritmo, es necesario saber cómo se va a implementar. Teniendo en cuenta la plataforma sobre la que se va a producir el desarrollo, habrá que definir cómo resolver cada una de sus partes, si en hardware o en software. Podemos verlo como un compromiso, el cual viene generalmente influenciado por la máxima que dice *que las tareas complejas algorítmicamente es mejor ejecutarlas en software ya que simplifica enormemente el diseño, mientras que las complejas computacionalmente en hardware, ya que su ejecución es mucho más rápida.*

Según eso, encontramos tres alternativas que coexisten en el dominio de la computación digital, y entre las que tendremos que decidirnos:

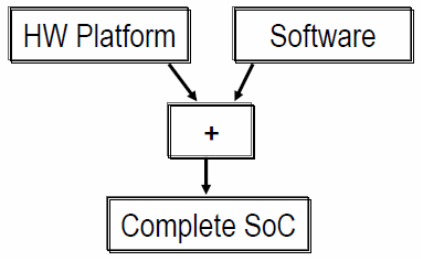

Ilustración 14. Metodología simplificada de desarrollo de un co-diseño HW-SW.

- Enfoque Hardware. El problema es resuelto mediante la implementación del algoritmo directamente en forma de puertas lógicas y/o circuitos integrados. Es el caso de una calculadora digital clásica, las que estamos acostumbrados a utilizar cuando estudiamos en el colegio. Estos diseños son específicos para ese problema concreto y una vez validados, suelen desarrollarse en forma de ASIC (Application Specific Integrated Circuit). Las ventajas de esta solución son, sobre todo, las relativas al tiempo de ejecución, es decir, la gran velocidad de respuesta del circuito o la seguridad de que va a concluir en un tiempo acotado. Sin embargo, como inconvenientes encontramos las relativas al coste, pues su desarrollo es lento y complicado y es necesario utilizar herramientas específicas y complejas, y las relativas a su nula versatilidad.

- Enfoque Software. Mediante la utilización de computadores de propósito general es posible llevar a acabo una implementación basada en algún lenguaje de programación, y ejecutarlo sobre aquél. Siguiendo esta alternativa, los diseños son más baratos, pues las herramientas son más sencillas, y requieren menos tiempo de desarrollo. Además, son fácilmente modificables, pues simplemente es necesario cambiar las líneas de código precisas, y volver a compilar. Los inconvenientes están relacionados sobre todo con el tiempo de ejecución, no acotado y generalmente varias órdenes de magnitud mayor al de los ASIC, aunque depende de la máquina en que se ejecute.

- Enfoque Hardware-Software. Implica la utilización de un microprocesador para tareas que requieran una solución más lenta o simplemente parametrizable o modificable, y el desarrollo de módulos ad-hoc hardware para el resto. Es necesario proporcionar una vía de comunicaciones para el intercambio de datos entre el procesador y los módulos, que generalmente suele ser en forma de bus. Para hacer la división hardware-software suele estudiarse cuáles serán las partes que más tiempo de proceso requieren y cuáles podrán ejecutarse concurrentemente. Este enfoque es actualmente muy utilizado, incluso en el hardware evolutivo [31], pues resulta flexible de cara a modificaciones futuras, requiere un menor esfuerzo en implementación que la solución hardware completa, ya que ambas sub-divisiones son más sencillas y se comporta mejor al ser

escalado. Este enfoque es el que siguen los SoC (System on Chip) como vemos en la Ilustración 14, muy en boga en la actualidad, y que buscan un compromiso de rendimiento/consumo que les obliga a elegir esta línea.

Más arriba definíamos el hardware reconfigurable como flexible. Tanto es así, que con la Virtex-II Pro con la que contamos sería posible realizar el diseño propuesto siguiendo cualquiera de los tres enfoques. En cualquiera de ellos resulta posible testear una y otra vez nuestro diseño sin necesidad de modificar la circuitería interna. Hemos elegido un co-diseño como solución (SoC), pues consideramos que es la que resulta más beneficiosa fijándonos en el compromiso entre ventajas e inconvenientes. Sobre todo, resulta muy flexible de cara a futuros cambios y queda muy bien estructurada. Varios módulos hardware ad-hoc son diseñados para la valoración de genotipos y para el procesado, partes que resultan rápidas y eficientes. Además, el hecho de poder implementar el algoritmo genético en un lenguaje como C y ejecutarlo sobre un procesador nos permite una fácil parametrización, imprescindible para obtener buenos resultados experimentales.

3.4. ARQUITECTURA DEL SISTEMA.

De cara a la comprensión del funcionamiento de un sistema de cierta envergadura, resulta de gran ayuda comenzar la exposición con la explicación de su arquitectura desde su visión más global e ir descendiendo hasta los detalles de la implementación de los módulos más característicos, consiguiendo de esta manera obviar detalles innecesarios. El enfoque que se va a seguir en este apartado es, entonces, top-down.

Tal y como se aprecia en la Ilustración 15, los bloques principales que conforman este SoC son:

- MicroBlaze. En su versión 7.10d. Se trata del módulo IP instanciable desde la herramienta EDK, y que contiene el código a sintetizar del soft-core de Xilinx. Se distingue en negro en la Ilustración 15 y se trata de uno de los elementos principales de la arquitectura. Como se puede ver queda asociado a través de dos interfaces de comunicación diferentes al bus PLB (Processor Local Bus) como maestro, y al bus de depuración como esclavo.

- Bus PLB v4.6. Este bus compartido de datos e instrucciones de 128, 64 o 32 bits es instanciable también como un core parametrizable. El IP en sí mismo constituye una parte básica del sistema, pues recoge las comunicaciones entre los diferentes componentes, como se puede apreciar en la Ilustración 15, dibujado en color beige. Provee la infraestructura para el intercambio de información entre múltiples maestros y esclavos. Esta constituido de un controlador, un temporizador y

de unidades de escritura y lectura de datos en espacios de direcciones disjuntos.

- MDM. Acrónimo de Microprocessor Debug Module. Se trata de un IP enfocado a la depuración de sistemas basados en uno o más microprocesadores MicroBlaze. Se conecta a la herramienta XMD, ya explicada más arriba, y entre ellos implementan una técnica de depuración basada en el estándar JTAG del IEEE. De cara a comunicarse con el resto de componentes implementa las interfaces con los buses PLB y de depuración, como esclavo y maestro respectivamente.

- Controlador y memoria BRAM. Como ya se explicó antes, la memoria de la Virtex II Pro se basa en módulos Block RAM distribuidos a lo largo y ancho del chip. Puesto que nuestro diseño consta de un microprocesador en el que se va a ejecutar una aplicación software, será necesario contar con algún tipo de soporte que contenga los datos e instrucciones en los que se basa dicho programa. Ese es el objetivo de la inclusión de esta memoria, y de su correspondiente controlador asociado. Su tamaño es de 64KB y se conecta al PLB como esclavo. Podemos ver su disposición a la derecha de la Ilustración 15.

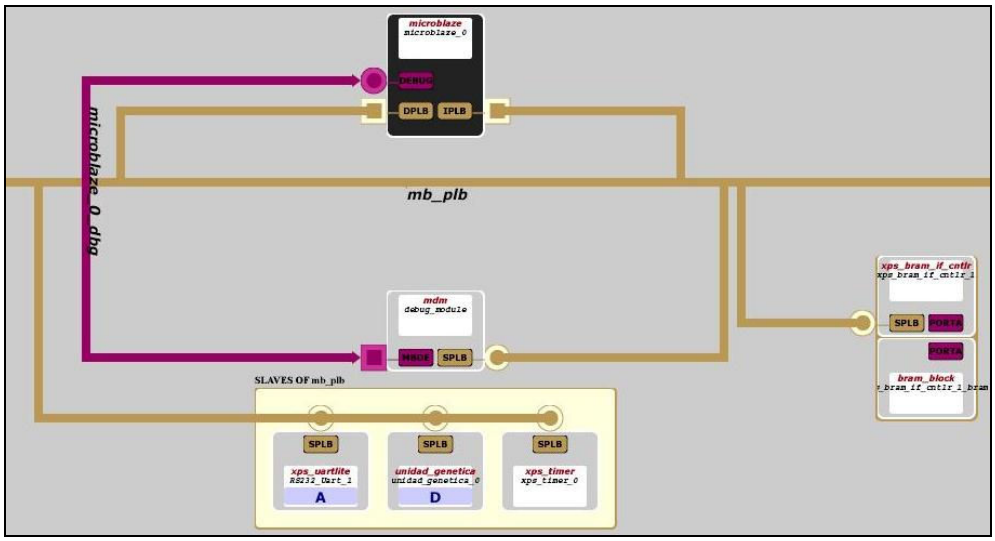

Ilustración 15. Diagrama de bloques de la arquitectura completa de nuestro sistema.

- Unidad asíncrona de comunicación serie. Se trata del módulo visible en la Ilustración 15 en la esquina inferior izquierda. Se dispone conectado al bus PLB y permite la comunicación mediante un protocolo serie RS-232 con el exterior, lo que nos permite imprimir y visualizar en la pantalla de nuestro PC información de las aplicaciones que se ejecutan en el procesador.

- Temporizador/Contador XPS. Se trata de otro módulo IP perteneciente a Xilinx y que implementa dos contadores de 32 bits accesibles por el desarrollador. Se conecta así mismo al bus PLB como esclavo, y en nuestro caso lo utilizamos para medir tiempos de ejecución entre fases del algoritmo evolutivo, y para introducir cambios "inesperados" tras cierto tiempo el sistema. Así, podríamos simular que tras cierto tiempo algunas partes del circuito quedan inactivas, y estudiar cómo la evolución busca soluciones obviando dichas celdas.

- Interfaz IPIF-IPIC. Siglas de Intellectual Property InterFace y de Intellectual Property InterConnect. Aunque no es un periférico en sí, y por ello no aparece como un componente más dibujado en la Ilustración 15, se trata de una parte fundamental para llevar a cabo la comunicación entre la lógica definida por el usuario y los módulos ya predefinidos. IPIF viene a ser un bloque intermedio entre el IP creado y el bus de comunicaciones e implementa una interfaz con éste, tomando el control de las señales, la gestión del protocolo de comunicación y demás aspectos. IPIC es la interfaz que el módulo definido por nosotros visualiza y con la que debe interactuar. La utilización de este estándar es opcional, pero su gran ventaja es la facilidad de uso y la portabilidad hacia otros sistemas con, por ejemplo, otros buses que también soporten IPIF-IPIC (OPB de Xilinx).

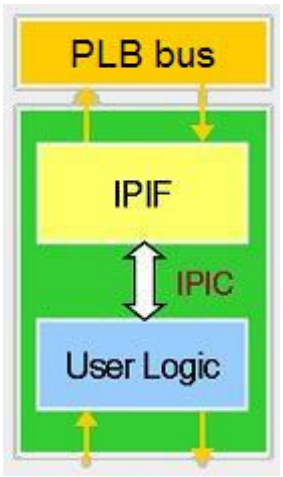

Ilustración 16. Interfaz de comunicación entre un módulo propio y el bus PLB.

- Unidad Genética. Se trata del último de los periféricos incluidos, y es el codificado por nosotros a lo largo de este trabajo de investigación. Queda conectado al bus PLB en modo esclavo mediante una interfaz IPIF-IPIC, que proporciona soporte y comunicación bi-direccional. Esta comunicación resulta clave pues es necesaria la sincronización con el procesador para el envío y

recepción de señales de control y para la recepción de nuevos cromosomas. El siguiente apartado explica con mayor detalle este componente.

3.5. MÓDULO EHW.

Una vez se tiene clara la arquitectura en su nivel de abstracción más alto, es necesario bajar de capa e introducirse en los entresijos del IP que se ha desarrollado.

Como ya se ha introducido al principio de esta memoria, uno de los objetivos que tiene el sistema que estamos especificando es que se trate de un enfoque ejemplar, y por ello debe quedar bien dividido modularmente. Esto beneficia de forma clara a su posterior reutilización y mejora. Para conseguirlo, hemos seguido el enfoque de los Componentes Evolutivos [14], [15], [16]. Para conseguir un diseño modular óptimo, se divide el sistema en un módulo Entorno y otro, Componente Evolutivo.

El primero, aparte de actuar como nodo maestro a la hora de comunicarse, es el encargado de comprobar si la funcionalidad del circuito cargado y ejecutándose cumple con las especificaciones dadas. Dependiendo de cómo se comporte para todas las posibles entradas, el sub-módulo Unidad Fitness devolverá una puntuación que denota su bondad. El control de todo el Entorno se lleva a cabo desde una máquina de estados dedicada (Controlador 2), que indica qué hacer en cada momento.

El Componente Evolutivo es, sin embargo, dónde se evolucionan los diferentes posibles circuitos y dónde se implementan. Así, se divide en una Unidad Genética y en un Circuito Reconfigurable. Todo ello queda dirigido por otro controlador propio (Controlador 1). Hemos elegido semejante distribución sobre todo por su modularidad, claridad y porque el Componente Evolutivo sirve como base para diferentes diseños evolutivos.

En la Ilustración 17 se puede visualizar un esquema gráfico de la división modular del IP desarrollado. En la parte alta vemos los dos componentes Entorno (Environment) y Componente Evolutivo (Evolvable Component) ya introducidos y la interfaz entre ellos. Como el punto de partida para la ejecución se encuentra en los botones de la placa de desarrollo (push buttons), sus señales entran al sistema a través del componente Entorno y será este el encargado de dirigir la ejecución del sistema. Su controlador es el definido como maestro y por tanto el del Componente Evolutivo será el esclavo, con lo que simplemente se encargará de replicar los mensajes del controlador maestro, pero dirigidos hacia sus módulos internos.

Siguiendo en la misma ilustración, en su parte inferior podemos ver un diagrama esquemático más detallado del la unidad genética. Como vemos, aparte de

su funcionalidad la cual será detallada más adelante, sirve de puente entre esta parte del sistema y la parte desarrollada en software, y que se ejecuta en el procesador. El bus de intercambio de datos es el ya nombrado PLB y las interfaces intermedias entre ellos IPIF-IPIC. También se ilustran los registros que actúan como buffers de entrada/salida en el proceso. A la vez, en la parte más inferior del diagrama y en un color azul más intenso, puede verse el pseudo-código del algoritmo genético que se ejecuta en el procesador.

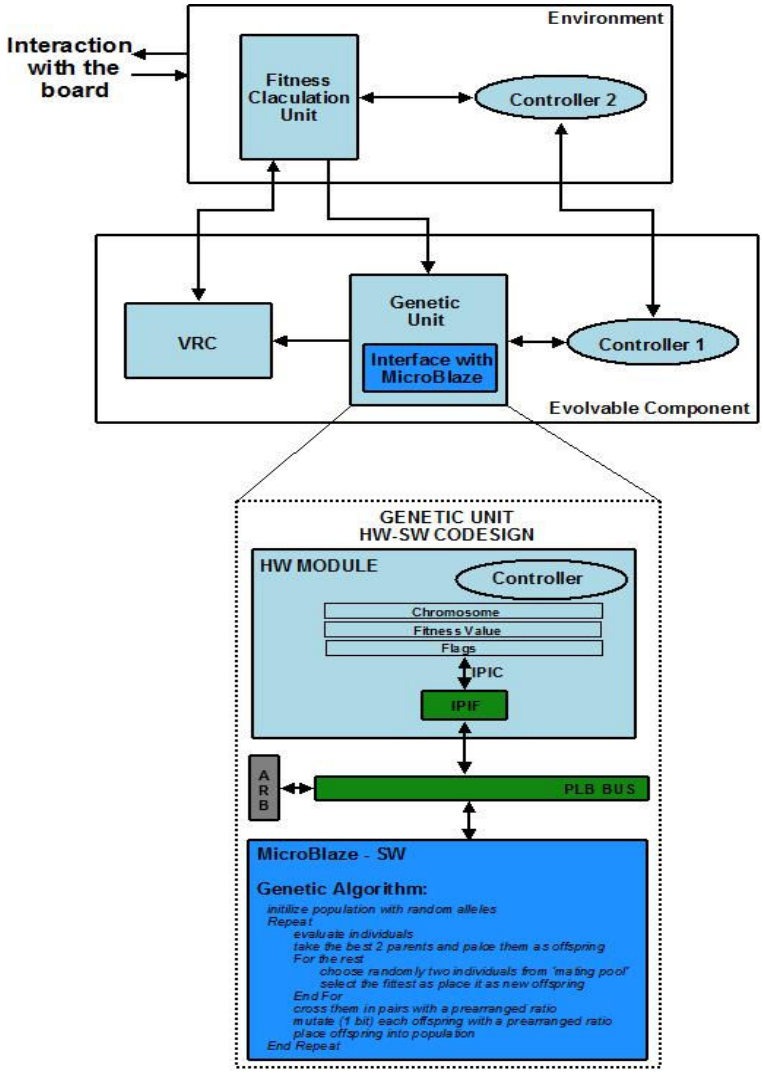

Ilustración 17. Diseño modular del sistema implementado. Se pueden observar los diferentes módulos hardware desarrollados en VHDL y el algoritmo evolutivo en C sobre el procesador.

3.5.1. MÓDULO ESPECÍFICO ENVIRONMENT.

La primera de las sub-divisiones en las que queda partido nuestro IP es el componente denominado Entorno. Se trata de la única parte del sistema que tiene conocimientos específicos de la aplicación que se desarrolla, y por ello contiene el módulo evaluador. Así mismo, el controlador principal queda contenido en ella, el cual interactúa a su vez con un supuesto operador responsable de los controles externos existentes en la placa de desarrollo.

3.5.1.1. Fitness Calculation Unit.

La Ilustración 18 es un punto de partida idóneo para la explicación de la funcionalidad de la unidad de cálculo del fitness de nuestro sistema. Como se puede observar consta de dos memorias o registros, un pequeño controlador y un contador como elementos esenciales. Al igual que el resto del sistema, ha sido desarrollado en VHDL, y simulado aisladamente para comprobar su funcionalidad.

Como ya se comentó más arriba, el componente Entorno es que el posee un conocimiento amplio de la funcionalidad del sistema y por ello el que puede evaluar si el circuito cargado en ese instante es de suficiente calidad o no. Debemos hacer un inciso en este punto y clarificar que con la expresión "suficiente calidad" queremos hacer hincapié en que para ciertos diseños de mucha complejidad podría ser que una solución sea factible mientras se consiga un, por ejemplo, 90% de sus salidas esperadas. Al tener nuestro diseño una funcionalidad más sencilla, sólo nos valdrán los circuitos que consigan el 100% de aciertos en sus salidas para toda la parrilla de posibles entradas.

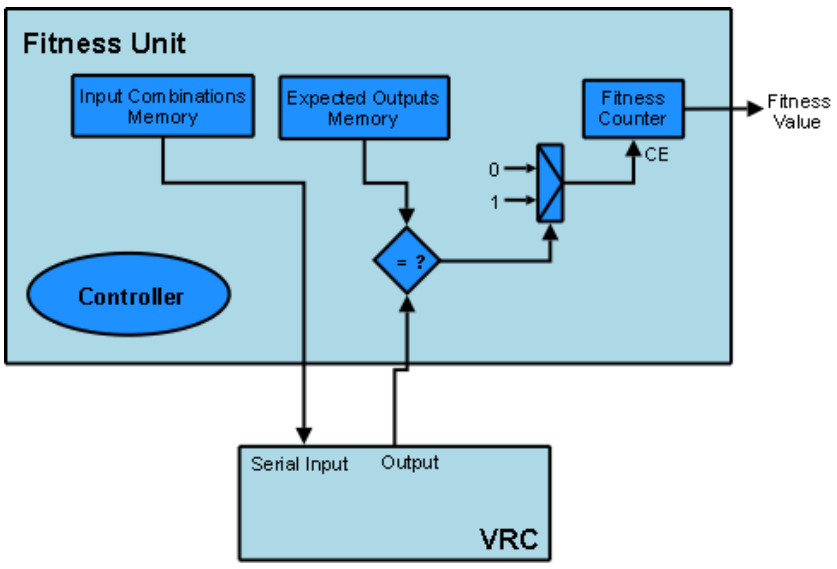

Ilustración 18. Esquema modular del módulo que calcula el fitness de un cromosoma.

Como este módulo es el encargado de validar una solución, resulta ser el cerebro de dicho proceso. Es por ello que proporciona las entradas y, comprueba si las salidas se corresponden con las que el reconocedor debería devolver. Así, contiene una memoria de configuraciones de entrada implementada en forma de un registro de desplazamiento en la cual se guarda una cadena binaria de comprobación que se envía de forma escalonada a la matriz de celdas reconfigurables definida por nosotros (VRC). La memoria que contiene las salidas correspondientes a dichas entradas queda también contenida en su interior, siendo igualmente un registro de desplazamiento. Éstas serán comparadas con las salidas reales del circuito. En caso de ser iguales, se incrementará el contador de Fitness en una unidad mediante la activación de la señal Count Enable; en caso contrario, dicha señal permanecerá desactivada y el contador mantendrá su estado.

La cadena binaria de comprobación debe contener todas las posibles combinaciones de entrada. Al tratarse de un reconocedor de tres bits, la cadena mínima completa de comprobación deberá pasar por las 8 posibles configuraciones, y contendrá por ello 10 bits: 0001011100. Es preciso especificar que hasta el tercer bit no se considera un patrón comparable y por ello son necesarios dos bits de entrada adicionales. En este caso, el rango de la función de fitness iría desde 0, situación en la que ninguna de las salidas ha sido la esperada, hasta 8, caso en el que hemos evolucionado un circuito correcto.

En cualquier caso, y para aumentar el rango de dicha función, hemos decidido aumentar esta cadena para que introduzca tres repeticiones de cada posible terna. Así, el registro de combinaciones de entrada será un registro de desplazamiento de 26 bits inicializado con la cadena: 00010111001110100001011100. En consecuencia, las salidas serán calculadas e introducidas en un registro de desplazamiento de 24 bits y el rango de la función de fitness variará entre 0 y 24, siendo este último valor el caso en que un circuito se comporta acorde con las especificaciones. El contador que lleva el control de la aptitud de la solución evaluada es genérico y de 5 bits, y al finalizar el proceso enviará su valor directamente a la unidad genética, que se encargará de transmitirlo a su vez hasta el procesador. En cualquier caso, merece la pena destacar que el diseño es lo suficientemente parametrizable como para cambiar estas características mediante la introducción de pequeños cambios al código.

En cuanto al controlador de la propia unidad de fitness, se trata de una pequeña máquina de estados de tipo Moore que mantiene el control del envío de entradas, la recepción y comparación de salidas y la cuenta de configuraciones correctas. Podríamos decir que funciona en modo esclavo respecto a l Controlador 2, del Entorno.

3.5.1.2. Controlador 2.

Esta máquina de Moore resulta ser una de las dos partes en que se divide el cerebro del sistema total. Su diagrama de estados queda descrito en la Ilustración 19. Una vez inicializado mediante el reset del sistema, este autómata se queda a la

espera de alguna señal que le haga entrar en uno de los tres modos de funcionamiento que tiene.

Por una parte, podría recibir la señal invalidateBest, que proviene de la misma placa, y descartar el mejor circuito evolucionado hasta el momento. Para ello, simplemente deriva esta orden al Controlador 1 mediante la activación de la señal invalidateBestFit. Con este modo de funcionamiento se pretende aumentar las posibilidades de la evolución y ordenar la búsqueda de otra solución si no se esta conforme con la actual.

El segundo de los modos corresponde con la carga del mejor de los circuitos evolucionado hasta ahora en la lógica programable del módulo VRC. Como vemos en el autómata, se llega al recibir la señal bestChrom, proveniente también del exterior tras la interactuación de un operador. Al llegar al estado 6, una señal de función similar llamada bestC es enviada al Controlador 1. Hasta que el circuito no quede cargado en el VRC y la unidad genética dé su aprobación, nuestro autómata estará esperando en el mismo estado. Una vez recibida esta señal, circuitPrep, el controlador dará el visto bueno y se quedará a la espera de una nueva orden.

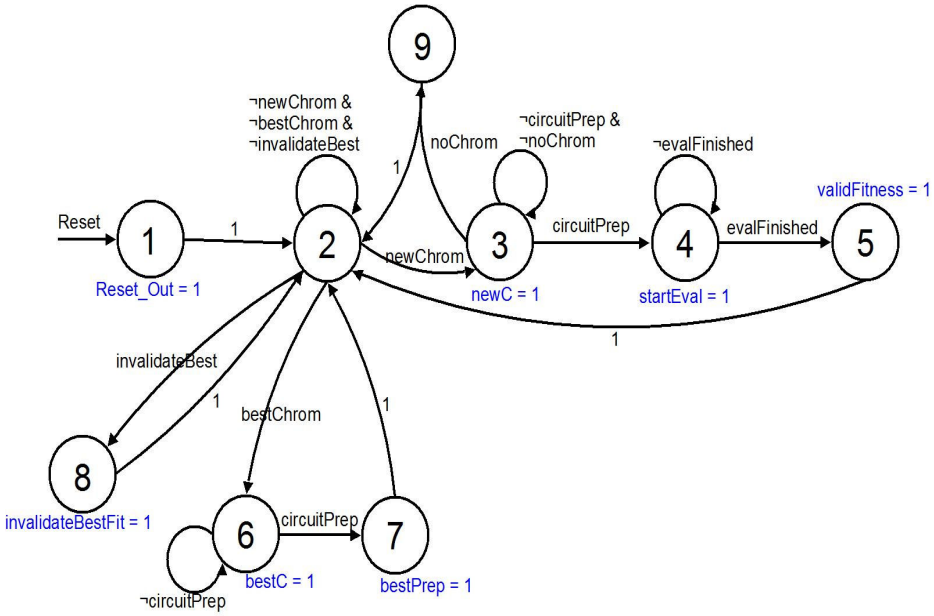

Ilustración 19. Máquina de estados tipo Moore correspondienteal Controlador 2, maestro del sistema.

El tercer y último modo de funcionamiento se inicia cuando al interactuar el operador pide al sistema la evolución de una nueva solución. La señal newChrom será recibida desde el exterior y entraremos en el estado 3, enviando al mismo tiempo una señal de petición de evolución al Controlador 1. El autómata se queda a la espera de respuesta, que podrá ser positiva, en forma de activación de la señal circuitPrep, o negativa, mediante la señal noChrom. En este último caso, la evolución ha llegado a su fin bien porque se ha llegado al límite de generaciones o bien porque

se ha encontrado una solución correcta. En cualquier caso volveríamos al modo espera. Por otra parte, si hemos recibido una notificación positiva significa que un cromosoma ha llegado a la unidad genética y que es posible empezar a evaluarlo. En este momento, se avisa a la unidad de cálculo del fitness de este hecho mediante la activación de startEval. Una vez empieza la evaluación, el autómata se queda a la espera de la señal que notifique la finalización de esta, evalFinished, proveniente del controlador de la unidad de fitness. Una vez la recibamos, sólo nos queda informar al componente evolutivo de este hecho mediante la señal validFitness, volviendo tras lo cual al estado de espera.

3.5.2. MÓDULO GENÉRICO EVOLVABLE COMPONENT.

Como ya se introdujo anteriormente, el componente evolutivo pretende ser la parte genérica del sistema. Aunque esto resulte incongruente hasta cierto punto con la propia naturaleza de un sistema de estas características, resulta reutilizable en diseños con especificaciones que no sean muy dispares.

Este componente queda dividido en tres partes, el Controlador 1, el Circuito Virtual Reconfigurable y la Unidad Genética. Los describimos con más detalle a continuación.

3.5.2.1. Virtual Reconfigurable Circuit (VRC).

La parte que quizás caracteriza en mayor medida a nuestro sistema es el módulo de celdas reconfigurables, VRC. Supone una arquitectura totalmente reconfigurable de forma dinámica y rápida por sí misma, con lo que es posible aplicar sobre ella las técnicas de hardware evolutivo. Como explicaremos en las siguientes líneas, el circuito se configura a partir de un bitstream o cadena binaria que corresponde directamente con el cromosoma evolucionado en el algoritmo genético. En nuestro caso, esta cadena esta compuesta por 186 bits, los cuales se distribuyen entre los multiplexores de enrutado de las diferentes celdas y los de selección de funcionalidad. Todo ello queda claramente plasmado en la Ilustración 20.

El circuito virtual reconfigurable esta compuesto por una matriz bidimensional de celdas programables en cuando a funcionalidad y enrutamiento de entrada. Se trata de 24 celdas dispuestas en cuatro filas y seis columnas. Cada una de las entradas de estas cajas se encuentra conectada bien a la salida de otra de la columna inmediatamente anterior, o bien a las entradas generales al sistema. Este enrutado se realiza a través de multiplexores que según una señal de selección deciden qué conexión es la seleccionada para dicha entrada del bloque funcional.

Las entradas generales del circuito virtual reconfigurable llegan en cada ciclo de ejecución, tratándose del patrón a reconocer y el bit de la cadena serie que corresponda. La salida total del circuito se obtiene al seleccionar una de las cuatro

salidas de las celdas de la última columna, es decir, que para ello contamos con un multiplexor de salida de 2 bits.

Las investigaciones consultadas sobre los VRC [14], [15], [16] suponían el desarrollo de sistemas combinacionales que en cada ciclo, independientemente del número de celdas internas, conseguían un valor de salida del VRC dependiente de sus entradas. Nuestro desarrollo es, sin embargo, diferente, pues estamos lidiando con un sistema secuencial, y el enfoque varía. En éstos es necesario considerar que el sistema recorrerá una serie de estados (y en consecuencia ciclos de ejecución) hasta encontrar el que dé la salida correspondiente. Aunque los estados en sí mismos no son predefinidos en esta solución, sí deben serlo los elementos de almacenamiento que conllevan, los biestables. Es por ello que cada celda contiene a la salida del bloque funcional un flip-flop por flanco de subida que guarda el valor de dicha salida un ciclo. De esta manera llegamos a la conclusión de que la salida correspondiente a las entradas introducidas en un ciclo dado será recogida seis ciclos más tarde, cada uno correspondiente a una columna del VRC. Además, por tratarse de un reconocedor de tres bits, es necesario permitir al sistema que recoja las entradas serie de tres ciclos consecutivos. Es por ello que mediante el aumento de un bit en la selección de entrada, las columnas segunda y tercera quedan conectadas, además de a su columna inmediatamente anterior, a las entradas generales. Con esto se permite al sistema recojer el valor de la entrada serie esos tres ciclos consecutivos, obteniendo su salida correspondiente seis ciclos después de la llegada de la primera de ellas.

La funcionalidad de cada una de las celdas viene impuesta por una terna binaria que decide entre las 9 posibles funciones que se muestran en el detalle de la parte inferior de la Ilustración 20. Como vemos, se trata de hardware evolutivo a nivel funcional. Con esta amplia gama de funciones se pretende aumentar la ambigüedad y con ella aumentar las posibles soluciones al sistema.

Este componente VRC ha sido desarrollado en VHDL mediante técnicas mixtas de desarrollo, es decir, combinando un enfoque funcional y un enfoque estructural. Siguiendo un esquema bottom-up, empezamos definiendo las celdas internas, y simulándolas, y pasamos a definir el VRC al completo.

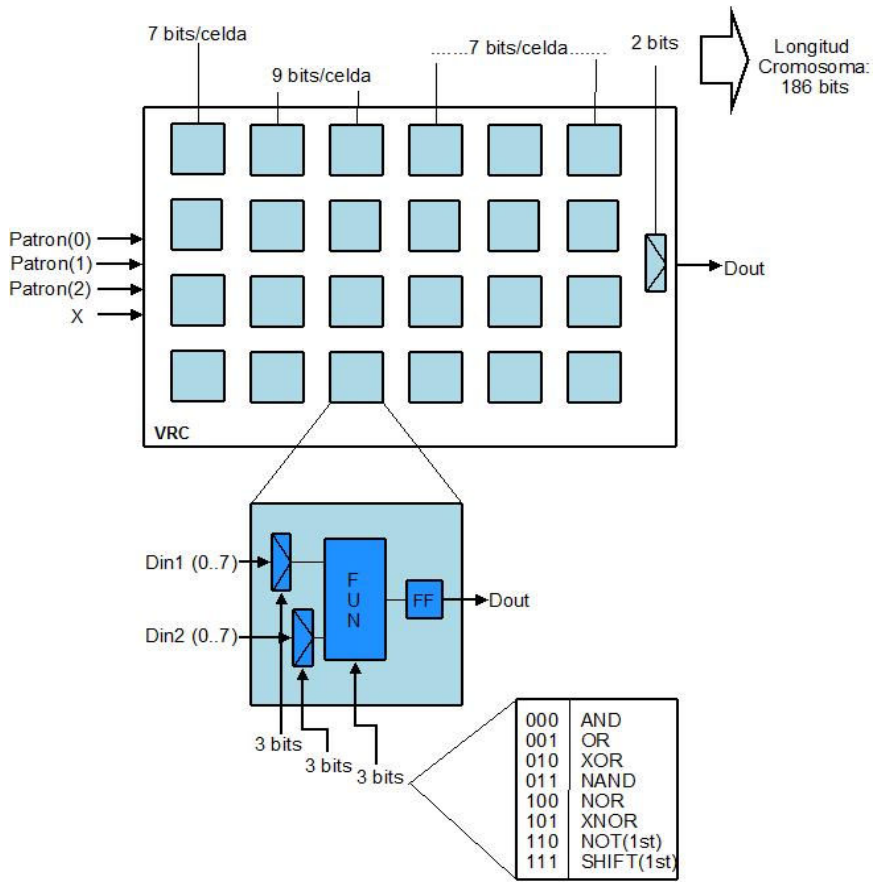

Ilustración 20. Esquema modular del circuito virtual reconfigurable (VRC), así como detalle de sus celdas internas.

3.5.2.2. Genetic Unit.

El subsistema más complejo del IP es, sin lugar a dudas, la Unidad Genética. Como ya se introdujo antes, el algoritmo genético se ejecuta sobre el Microblaze, pero los cromosomas que éste genera deben ser validados en el propio IP. Para su envío, hacemos uso del bus PLB, el cuál además nos proporciona una interfaz genérica para el envío y recepción de datos. Ya se explicó más arriba que ésta se llama IPIF-IPIC, y al incluirla en nuestro IP, se encarga de gestionar la comunicación de E/S. Es por ello que el módulo que ahora nos ocupa hace uso de ella. Ésta comunicación se basa en el mapeado de memoria, es decir, que ciertas posiciones de ésta van a estar dedicadas al intercambio de información entre emisor y receptor. En nuestro diseño, la comunicación será bi-direccional y llevada a cabo entre el procesador y el módulo IP Unidad Genética.

Como se observa en la Ilustración 17, ha resultado necesaria la inclusión de varios registros para el intercambio de información. En ella aparecen tres rectángulos que representan los registros utilizados para enviar cromosomas del procesador

hacia el IP, otro para enviar el valor de la aptitud de un circuito en éste de vuelta al procesador, y un tercero de flags, es decir, de control.

Al trabajar el bus PLB y al estar la interfaz IPIC-IPIF desarrollados para 32 bits, los registros de comunicación que se implementan son también de este tamaño. Para el primero de ellos, el dedicado al cromosoma, es necesario en realidad la utilización de 6 registros, ya que debemos permitir el envío de 186 bits. Aunque el envío de cada uno de los paquetes de 32 bits se realice secuencialmente en el bus, se ha decidido utilizar más memoria de cara a simplificar el sistema. El segundo de los registros de comunicación, el de fitness, utiliza en realidad sólo 5 bits. En cualquier caso, y al tratarse de una información en sentido contrario, queda incluida en un registro de 32 bits dedicado. Por último, encontramos un registro dedicado a mantener una correspondencia a nivel de estados entre el procesador y el IP.

Como vemos en la Ilustración 21, el registro de flags queda conformado por 11 posiciones que plasman el intercambio de señales de control entre el procesador y el IP. Éstas se encuentran coloreadas. Las que se muestran en color azul corresponden a peticiones que el IP envía al MicroBlaze, mientras que las dibujadas en verde corresponden a acciones de confirmación que el procesador devuelve al módulo HW. De esta manera se lleva a cabo una comunicación que viene inducida por los autómatas de los Controladores 1 y 2. De hecho, si nos fijamos con más detalle, veremos que estos flags tienen una correspondencia directa con las transiciones en aquellos. Profundizando un poco más, se explica a continuación el significado de dichos flags brevemente:

- flagProcReset. Petición del IP hacia el procesador para que reinicie los parámetros de la evolución.
- flagNewChrom. Petición que exige al procesador un nuevo cromosoma.
- flagSetFitness. Una vez evaluado un circuito, el IP indica al procesador que el valor obtenido con la función de aptitud se encuentra guardado y listo para el envío en el registro utilizado para dicho fin.
- flagBestChrom. Petición al procesador para que el próximo cromosoma que envíe sea el mejor evolucionado hasta el momento.
- flagClrBestFitness. Al contrario, petición para que el procesador borre el mejor cromosoma obtenido hasta el momento.

En sentido contrario, también hay una serie de flags que merece la pena explicar para la comprensión del sistema al completo:

- flagProcReady. Una vez recibida la petición de reinicialización de la evolución, y llevada a cabo ésta, el procesador comunica al IP dicho hito.
- flagNewChromEnMem. El procesador indica al IP que ya ha enviado un nuevo cromosoma a través del bus de datos.

- flagFinEvolMem. El procesador indica al IP que puede servirle un nuevo cromosoma puesto que la evolución ha concluido. Esto puede deberse a que se ha evolucionado un circuito acorde con las especificaciones o a que hemos alcanzado sin éxito el límite máximo de evoluciones acordado de antemano.
- flagBestChromEnMem. Notifica al IP que el mejor cromosoma evolucionado hasta el momento se encuentra listo en su espacio de recepción.
- flagBestFitnessClrEnMem. Por último, y tras el borrado del mejor cromosoma evolucionado hasta el momento, el procesador simplemente notifica al IP dicho hecho.

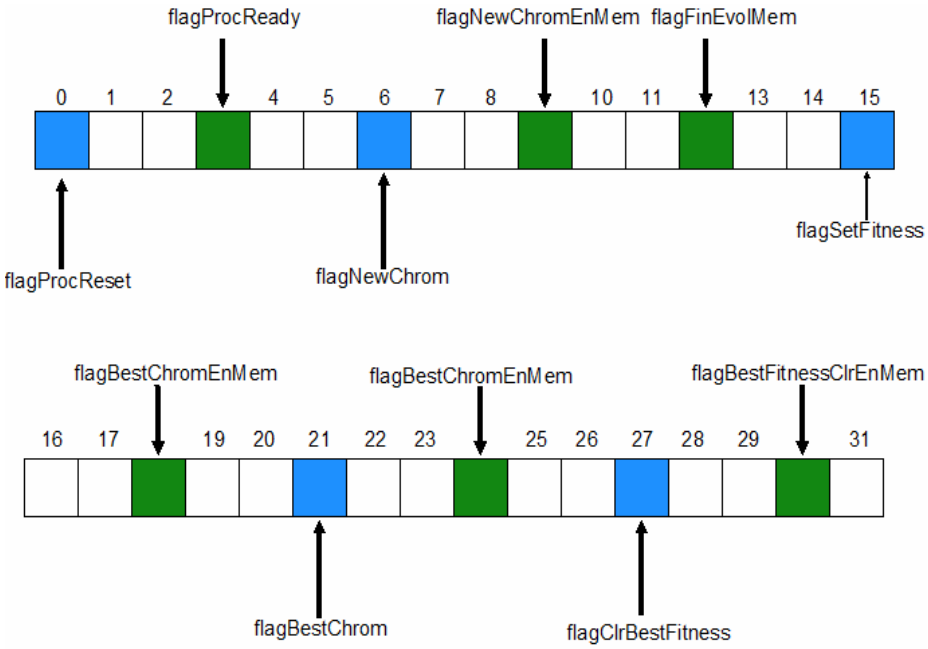

Ilustración 21. Esquema del registro utilizado en el IP para la comunicación de las señales de control entre éste y el procesador.

Respecto al resto de componentes y funcionalidades que la Unidad Genética proporciona al sistema, cabe destacar que lleva implementado un pequeño autómata de tipo Moore que se encarga de las comunicaciones con el procesador, así como de la preparación del VRC, pues se encuentra unido directamente a éste. Para una mejor comprensión de la disposición del módulo respecto al resto del sistema, insto al lector a volver la vista a la Ilustración 17.

3.5.2.3. Controlador 1.

El segundo de los controladores que forman el núcleo de control del sistema es el que se encuentra contenido en el Componente Evolutivo del IP, y que funciona en modo esclavo. Aunque los modos de funcionamiento que tiene vienen directamente

derivados del Controlador 2, merece la pena entrar en detalle para clarificar su funcionalidad. El diagrama de estados que modela su funcionamiento se expone en la Ilustración 22.

Una vez que se activa la función de reinicialización general mediante la señal Reset, el Controlador 1 va a permanecer estancado en el estado 2 a la espera de que la Unidad Genética, es decir, el módulo que se comunica con el procesador y por ende éste mismo, se encuentren listos para comenzar la normal ejecución. Una vez recibe la señal GU_ready, la máquina de estados entra en un estado estacionario a la espera de recibir alguna señal que le dirija hacia cualquiera de los tres modos básicos de funcionamiento. Se trata del modo en el que se hará una petición de evolución de un nuevo cromosoma, el modo en el que se pedirá la supresión del mejor de los generados hasta ahora, y el modo en el que se pide la carga del mejor de ellos.

Profundizando algo más en las transiciones, el autómata entrará en el modo de evolución una vez reciba una señal de petición de un nuevo cromosoma. Ésta se llama newC y viene del Controlador 2. Al realizar la transición al estado 4, activamos la señal NC que indica a la Unidad Genética que el procesador debe proporcionar una nueva cadena binaria. Así, puede ser que el resultado sea positivo y nos la devuelva, activando cFin, o negativo y que la evolución haya concluido, activando finEvol. En caso afirmativo la máquina de estado notificará al Controlador 2 que el circuito ya se encuentra preparado en el VRC y que por ello puede empezar a evaluar (circuitPrep). Una vez que el circuito se haya evaluado y contemos con su valoración, la señal validFitness se activará con lo que simplemente queda notificar lo propio a la Unidad de Fitness y volver al estado estacionario.

Los modos de funcionamiento que invalidan el mejor de los cromosomas evolucionados hasta el momento y el que pide cargar en el VRC el mejor de ellos para su posterior ejecución siguen un patrón de funcionamiento similar al del Controlador 2. Por ello, instamos al lector a retroceder unas páginas y releer la información relativa a dicha máquina de estados para clarificar las posibles dudas que pudieran surgirle al observar la Ilustración 22.

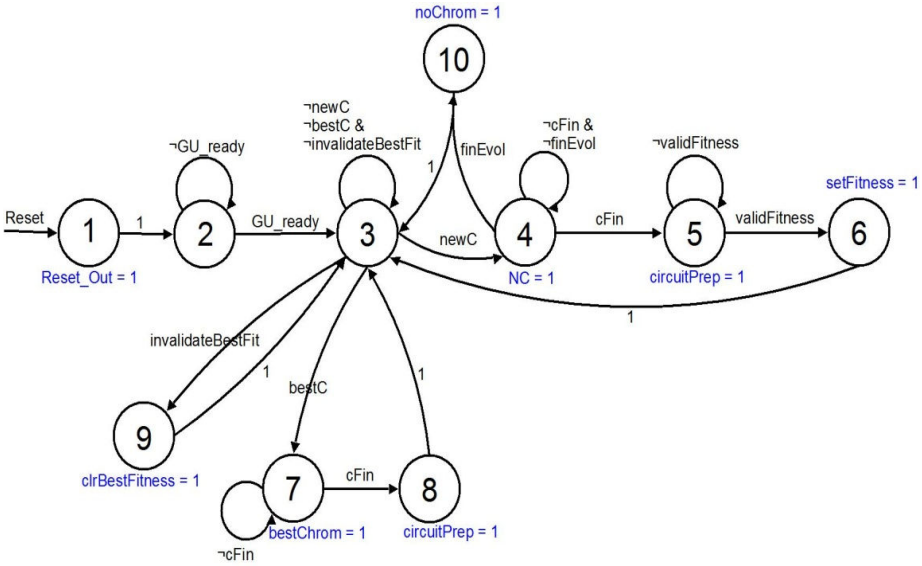

Ilustración 22. Máquina de estados tipo Moore que modela el funcionamientodel Controlador 1 (esclavo).

3.6. TÉCNICAS EVOLUTIVAS.

Una de las partes más interesantes del sistema es la encargada de ir evolucionando potenciales soluciones que serán validadas más tarde en el sistema. Como ya se introdujo en la primera parte de esta memoria, es prácticamente imposible llevar a cabo la evolución y posterior reconfiguración al nivel del bitstream que se carga en la FPGA, debido sobre todo a la peligrosidad que ello conlleva. La plataforma reconfigurable podría resultar dañada al introducir cadenas aleatorias incorrectas [17].

Para evitar ese tipo de problemas se trabaja en base a arquitectura reconfigurable que es reprogramable fácil y rápidamente. Sus cromosomas son caracterizaciones inequívocas de circuitos que utilizan sus elementos de enrutado y funcionales, y se basan en la Programación Genética Cartesiana también introducida anteriormente.

El esquema evolutivo que se lleva a cabo en el sistema queda caracterizado por un proceso como el mostrado en la Ilustración 23. Como se puede apreciar, a partir de una población inicial aleatoria, se inicia un ciclo evolutivo caracterizado por la validación del circuito en la FPGA, su valoración según la función de aptitud, la selección de los mejores individuos y la variación de ellos de cara a recomenzar el proceso. Esta variación se lleva a cabo mediante las técnicas de mutación y cruce,

aunque debido a la naturaleza del sistema y como ya se ha explicado con anterioridad, el operador mutación conlleva mejores resultados.

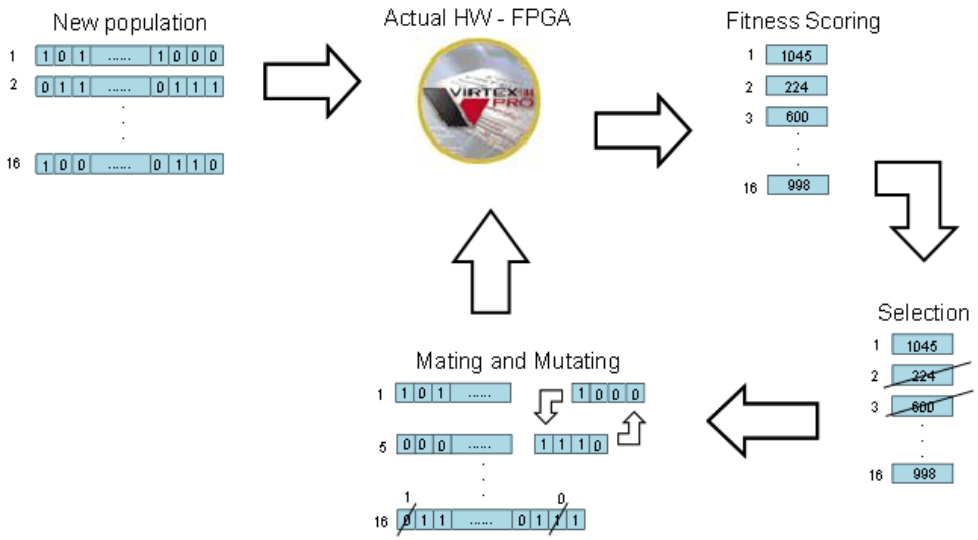

Ilustración 23. Esquema del proceso evolutivo aplicado a nuestro diseño.

La parte dedicada a la evolución de individuos en el sistema corresponde con la subdivisión software del mismo. Es decir, este proceso va a ser llevado a cabo en el procesador MicroBlaze empotrado. Mediante un algoritmo propio desarrollado en C, gestionaremos la comunicación con el IP, e iremos proporcionando a éste individuos según los vaya necesitando. De cara a manejar todo el proceso, es necesario implementar una solución que tenga en cuenta los flags de comunicación que se comparten con el IP. De hecho, la solución contempla un bucle infinito que escucha el registro de flags proveniente del IP, y en caso de que alguno haya sido modificado, actúa en consecuencia. Según esto, hay que dar respuesta a la petición de reinicialización, del mejor cromosoma generado hasta el momento, al borrado del mismo, a la grabación en memoria de la valoración del individuo evaluado y a la petición de uno nuevo.

Por razones de simplificación y puesto que el proceso evolutivo se lleva a cabo en el soft-core empotrado, la población de individuos y sus respectivas valoraciones son guardadas en memoria y accedidas desde el propio procesador. El código que gestiona este proceso es altamente parametrizable de cara a poder experimentar comportamientos bajo diferentes condiciones. Así, los tamaños de la población, la longitud del cromosoma, el ratio de mutación y cruce, el número máximo de generaciones o el límite que se impone al elitismo son parámetros fácilmente modificables. Más adelante, ya en el apartado de resultados experimentales, hacemos más hincapié en este sentido.

Entrando de lleno en el proceso de evolución de individuos, el algoritmo evolutivo que implementamos esta basado en el pseudo-código que se muestra en la

Ilustración 24. Éste a su vez se basa en las ideas expuestas en [44]. Como se aprecia en la figura, el proceso comienza inicializando la población de individuos de forma aleatoria. A continuación, se entra en un bucle en el cual se evalúan los individuos, y una vez valorados, entramos en la etapa de selección. En esta se comienza llevando a cabo la técnica conocida como elitismo, y mediante la cual se escogen los mejores individuos para que formen parte de la siguiente generación. Para el resto de futuros cromosomas, su material genético es elegido primeramente y copiado del individuo que mejor aptitud tenga entre dos seleccionados al azar. Más tarde, se pasa a la etapa de cruce y mutación.

```
initialize population with random alleles
Repeat
    evaluate individuals
    take the best 2 parents and place them as offsprings
    For the rest
        choose randomly two individuals from 'mating pool'
        select the fittest and place it as new offspring
    End For
    cross them in pairs with a pre-arranged ratio
    mutate (1 bit) each offspring with a pre-arranged ratio
    place offspring into population
End Repeat
```

Ilustración 24. Pseudo-código del algoritmo evolutivo de evolución de individuos.

El cruce y la mutación se ejecutan en el sistema de forma que la modificación de sus ratios de ejecución sea sencilla. Es por ello que no merece la pena entrar en detalles en cuanto a número de dichas variaciones, pero sí en cuanto a su técnica. El caso de la mutación se lleva a cabo sobre un solo individuo, y aleatoriamente se le modifica un bit. Quedando este invertido, el material genético del cromosoma habrá cambiado y puede que el comportamiento del circuito en consonancia. En cuanto a la variación, se lleva a cabo según un ratio, y por parejas de individuos consecutivos que forman la población. Cortando ambos miembros por un punto, sus materiales genéticos serán intercambiados y los nuevos vástagos unidos a la población de la siguiente generación. El orden en que se llevan a cabo estas funciones se muestra igualmente en la Ilustración 24.

4. RESULTADOS EXPERIMENTALES

Una vez explicados al detalle los fundamentos y teorías en los que se basa nuestra investigación, y descritos los entresijos de nuestro sistema, sólo queda informar sobre el funcionamiento del mismo mediante resultados experimentales. Así, en este apartado intentaremos exponer con la mayor rigurosidad y claridad posible dichos resultados, de manera que quede clara la utilidad, el objetivo y los logros alcanzados durante la ejecución. Comenzaremos por mostrar un resumen de los informes generados por la herramienta de síntesis utilizada, EDK, que demuestran los porcentajes de utilización y número absoluto de componentes empleados tanto por el sistema completo como por nuestro IP en concreto. Además, mostraremos una serie de soluciones de diseño evolucionadas por nuestra herramienta durante las diferentes ejecuciones llevadas a cabo. Por último, mostramos los diseños de un reconocedor desarrollado por la herramienta y que, suponiendo la detección de errores en varias celdas funcionales, es modificado para evitar la conexión de dichos elementos.

Empezamos analizando las estadísticas de utilización de lógica reconfigurable por parte de nuestro sistema. En el informe de diseño expuesto a continuación, vemos la cantidad de lógica necesaria que ha hecho falta para la síntesis del sistema al completo. En él incluimos:

```
Design Summary:

Logic Utilization:
  Total Number Slice Registers:         2,478 out of 27,392   9%
       Number used as Flip Flops:       2,477
       Number used as Latches:          1
  Number of 4 input LUTs:               3,219 out of 27,392   11%

Logic Distribution:
  Number of occupied Slices:            2,835 out of 13,696   20%
       Number of Slices containing only related logic:
                                        2,835 out of 2,835    100%
       Number of Slices containing unrelated logic:
                                        0 out of 2,835        0%
  Total Number of 4 input LUTs:         3,307 out of 27,392   12%
       Number used as logic:            2,770
       Number used as a route-thru:     88
       Number used for Dual Port RAMs:  384
       (Two LUTs used per Dual Port RAM)
       Number used as Shift registers:  65
  Number of bonded IOBs:                22 out of 556         3%
       IOB Flip Flops:                  5
  Number of RAMB16s:                    32 out of 136         23%
  Number of MULT18X18s:                 3 out of 136          2%
```

- El procesador MicroBlaze en su versión 7.10d.
- El módulo IP Hardware_Evolutivo, creado al completo por nosotros.
- El IP genérico xps_timer incluido en el EDK. Es un contador de 32 que suele ser utilizado para medición de tiempos y retardos.
- El bus de comunicaciones plb_4.6v.

- La unidad de comunicaciones serie RS232_UART que viene con la herramienta.
- La memoria BRAM y su controlador.
- El generador de señales de reloj.
- El modulo de depuración del procesador necesario para utilizar XMD con el MicroBlaze.

Como se aprecia en el informe, aun con todo ello instanciado llegamos a utilizar solamente una quinta parte de los recursos que proporciona nuestro dispositivo reconfigurable (20%). Como vemos, la Virtex-II Pro cuenta con 27.392 elementos de almacenamiento configurables, y de todos ellos hemos utilizado apenas el 10%. Algo parecido sucede con los bloques dedicados al cálculo de funciones, ya que únicamente el 11% de las LUT (LookUp Tables) son inferidas.

El informe temporal generado por la misma herramienta nos da idea de a qué frecuencia de trabajo máxima podría llegar a trabajar nuestro sistema. Aunque cada único de los módulos que lo componen puede trabajar a mayor frecuencia, el sistema solo podrá llegar hasta los 101MHz. Lo vemos en el informe copiado más abajo.

```
Timing summary:
---------------

Constraints cover 357002 paths, 0 nets, and 20783 connections

Design statistics:
       Minimum period:   9.842ns (Maximum frequency: 101.605MHz)
```

En cuanto a la unidad creada por nosotros, la unidad Hardware_Reconfigurable, vemos que los datos de síntesis se corresponden con lo diseñado. Así, en el siguiente cuadro de texto se muestran los resúmenes de síntesis de las celdas de la unidad VRC por separado, por ser las más características, tanto en su configuración de selección de entrada de 3 bits (VRC_Cell) como en la de dos bits (VRC_Cell_Ini). Para cada una de ella se infieren 3 multiplexores y un flip-flop de tipo D, como presumimos.

```
Synthesizing Unit <VRC_Cell>.
    Found 1-bit register for signal <Dout>.
    Found 1-bit 8-to-1 multiplexer for signal <op1>.
    Found 1-bit 8-to-1 multiplexer for signal <op2>.
    Found 1-bit 8-to-1 multiplexer for signal <sal>.
    Found 1-bit xor2 for signal <sal$xor0000> created at line 84.
    Summary:
       inferred   1 D-type flip-flop(s).
       inferred   3 Multiplexer(s).
Unit <VRC_Cell> synthesized.

Synthesizing Unit <VRC_Ini_Cell>.
    Found 1-bit register for signal <Dout>.
    Found 1-bit 4-to-1 multiplexer for signal <op1>.
    Found 1-bit 4-to-1 multiplexer for signal <op2>.
    Found 1-bit 8-to-1 multiplexer for signal <sal>.
    Found 1-bit xor2 for signal <sal$xor0000> created at line 76.
    Summary:
       inferred   1 D-type flip-flop(s).
       inferred   3 Multiplexer(s).
Unit <VRC_Ini_Cell> synthesized.
```

Si nos centramos en los informes de utilización de elementos de la placa para nuestro módulo (más abajo), vemos que son necesarios en total 520 slices de los 13696 con los que cuenta la FPGA. Puesto que cada uno de ellos cuenta con dos elementos de almacenamiento y posibilita la inferencia de dos LUT de 4 bits, los 531 biestables activados por flanco y las 701 LUT tendrán cabida en ese conjunto. Además, y como vemos, serán necesarios 212 bloques de E/S que actúen como interfaz de comunicación para el envío o recepción de señales con el exterior.

Teniendo en cuenta que el porcentaje de utilización del sistema, incluyendo todas sus partes, ascendía al 20% de los slices disponibles y que nuestro módulo en concreto no llega al 3% del total, podemos asumir que si no cambiamos la arquitectura podríamos replicar la solución más de 25 veces sin llegar a llenar el dispositivo. O lo que quizás fuese más interesante, proceder a escalar nuestro sistema en semejante proporción para da cabida a soluciones más complejas.

```
Device utilization summary:
---------------------------

Selected Device : 2vp30ff896-7

Number of Slices:                        520 out of 13696     3%
Number of Slice Flip Flops:              531 out of 27392     1%
Number of 4 input LUTs:                  701 out of 27392     2%
Number of IOs:                           212
Number of bonded IOBs:                   0 out of 556         0%
```

En cuanto al análisis temporal y como ya avanzamos antes, este módulo por separado sería capaz de trabajar a mayor velocidad, pues tal y como nos muestra la herramienta de síntesis el período mínimo de reloj alcanza los 4.213ns y por ello la frecuencia máxima llega hasta los 237.350MHz. Estos datos podemos verlos en la tabla resumen de más abajo.

```
Timing Summary:
---------------
Speed Grade: -7


Minimum period: 4.213ns (Maximum Frequency:
237.350MHz)
```

Una vez comentados los informes de utilización e inferencia de los componentes en la placa con la que contamos, vamos a entrar de lleno a tratar el objetivo de la investigación, la evolución de circuitos electrónicos digitales, y en concreto, el de nuestro reconocedor.

La primera de las pruebas realizadas es una comparación de cómo sería un reconocedor de patrones de tres bits que un diseñador desarrollaría al utilizar una arquitectura como la nuestra, y cómo son las soluciones reales obtenidas tras el procesamiento de nuestro sistema. Empezamos diseñando por nuestra cuenta un reconocedor para el patrón $P_2P_1P_0$=010. El diagrama correspondiente a esta situación se muestra en la Ilustración 25, y corresponde a la implementación directa del min-término que activa la salida de la función. Solo en el caso de recibir por la entrada serie, X, el valor $X_{t-2}X_{t-1}X_t$=010 la salida se activará. Es por ello que hemos utilizado dos puertas Nor y una And para la validación cada uno de los bits por separado, y las dos puertas And finales para la comprobación de que los tres bits cumplen dicha condición. Los biestables inferidos son necesarios para guardar el valor de la función

y de sus partes durante los seis ciclos que necesitan para recorrer la matriz virtual configurable. Es necesario aclarar que la correspondencia del patrón con la llegada temporal de la entrada serie ha sido decidida de manera arbitraria y es la siguiente: P_2-X_t, P_1-X_{t-1} y P_0-X_{t-2}.

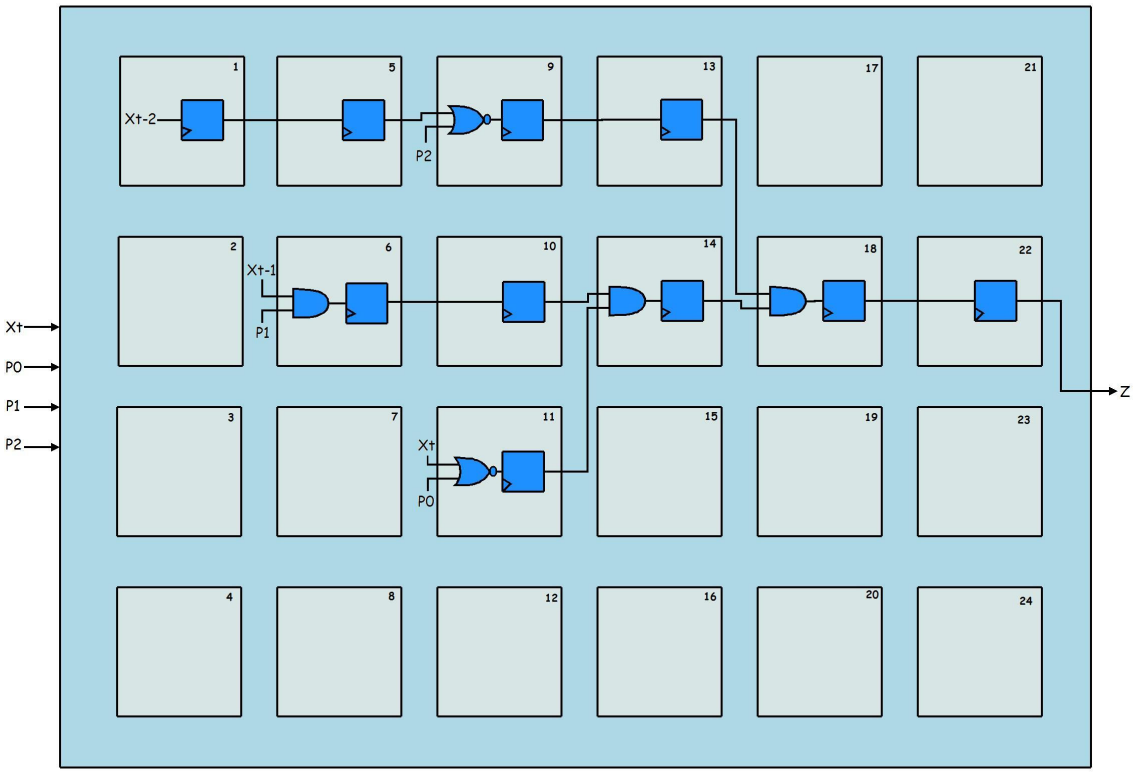

Ilustración 25. Posible diseño de un reconocedor de 3 bits llevado a cabo por un humano. (Patrón a reconocer: $P_2P_1P_0$=010).

Una vez contamos con un modelo con el que comparar, nos lanzamos a evolucionar posibles soluciones, las cuales serán prototipos validados que probablemente a un diseñador corriente o a una herramienta de síntesis no se les ocurriría desarrollar, ya que no tienen reglas predefinidas para llegas a dichas soluciones. Este es el ejemplo que encontramos en la Ilustración 26, que tratándose también de un reconocedor para el mismo patrón, infiere una solución bastante extraña para un humano. Aunque la biblioteca de funciones que pueden utilizarse son ocho, tal y como vimos en la Ilustración 20, en este ejemplo solo se usan cinco tipos. Hay que recordar que la elección de ellas se basa en el cromosoma, el cual no deja de ser aleatorio y por ello no se puede asegurar ni restringir en ningún momento el desarrollo con un tipo determinado de puertas ni con un límite de celdas lógicas.

En las figuras siguientes, Ilustración 27 e Ilustración 28, vemos dos nuevos diseños de reconocedores también evolucionados con nuestro sistema. Podemos apreciar que el número de celdas reconfigurables utilizadas también supera en

número al primer desarrollo, el inferido por nosotros manualmente. Éste no tiene por qué suponer la situación óptima en cuanto a recursos, pues no llegan a explotarse funciones potentes como la Xnor, por no resultar tan claras para el ser humano. Lo que sí aseguramos es que resulta ser la solución más comprensible. Teniendo eso en cuenta, podemos comparar y concluir que en cuanto a recursos:

- Solución 1 (manual): 10 celdas.
- Solución 2 (evolucionada): 9 celdas. 10% de mejora.
- Solución 3 (evolucionada): 12 celdas. Penalización del 20%.
- Solución 4 (evolucionada): 10 celdas. Mejoría del 0%.

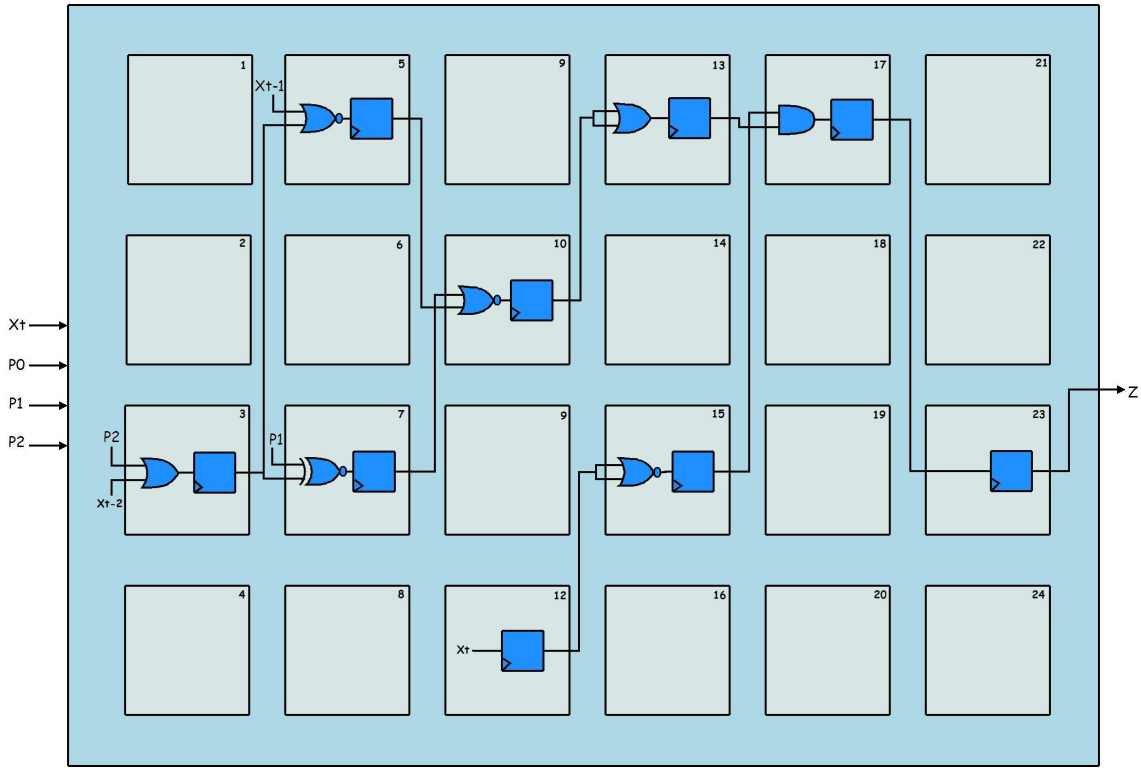

Ilustración 26.Primera alternativa de diseño evolucionada por el sistema, para un reconocedor de patrones de la cadena $P_2P_1P_0$=010.

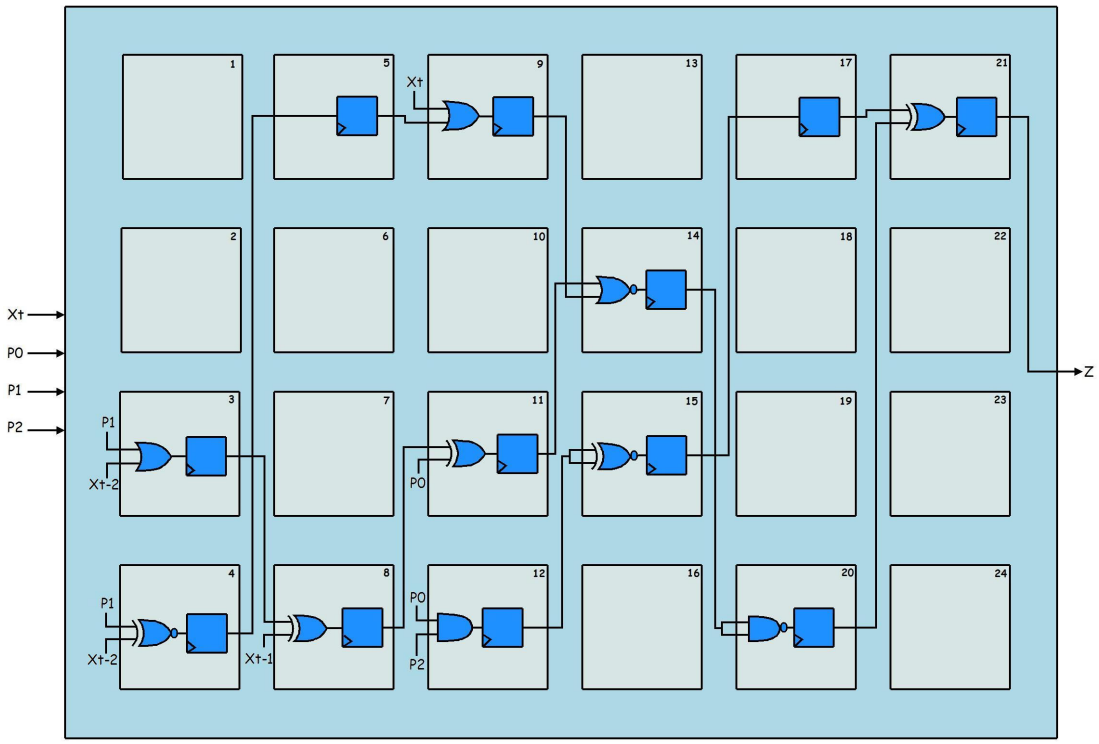

Ilustración 27. Segunda alternativa de diseño de un reconocedor del patrón 010.

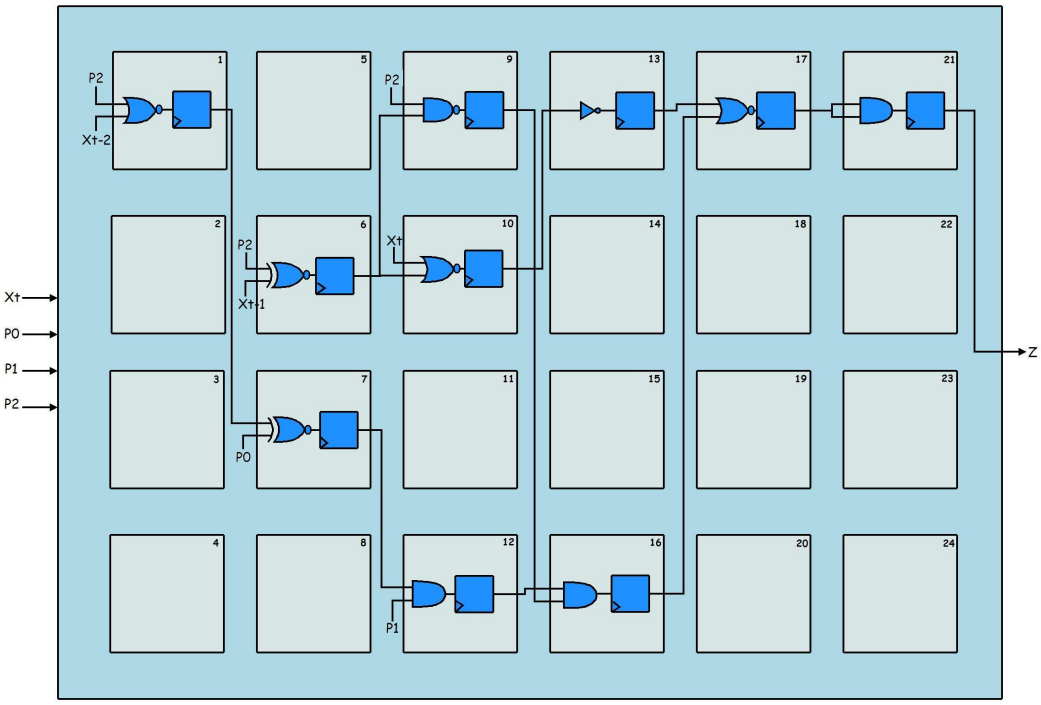

Ilustración 28. Tercera alternativa de diseño de un reconocedor de patrones, con $P_2P_1P_0$=010.

Concluimos que para este ejemplo y con sólo tres comparaciones ya notamos una tendencia hacia el consumo de unos pocos recursos más de los necesarios. De hecho, la penalización asciende al 10% en promedio. No deja de ser un compromiso a asumir el hecho de que nos importe empeorar la solución en área siempre y cuando el diseño se realice de forma automática.

Aumentando la complejidad del sistema, se supone a continuación que durante la evolución de posibles soluciones es posible que se produzcan fallos en las celdas que lleven a dichos componentes a la inutilización. Así, el sistema deberá tener dicha restricción dinámica en cuenta y cambiar el sentido de su búsqueda. Será exitoso en caso de que encuentre un circuito reconocedor que evada dichas celdas y configure la solución entre las restantes.

Esta nueva característica la llevamos a cabo modificando un poco el diseño. A partir de ahora suponemos que la configuración 111 de las celdas funcionales corresponde al caso descrito, y por tanto su salida resultará indefinida. Estos errores los podemos introducir al sistema contemplando dos magnitudes, tiempo de uso o utilización del circuito. Mediante la utilización del temporizador sintetizado podríamos simular el primer caso, considerando que tras un período de utilización el circuito comenzará a fallar. El segundo mediante la suposición de que tras el uso intenso del sistema este fallará en algún momento. Este uso podría caracterizarse según un número arbitrario de generaciones transcurridas en la evolución.

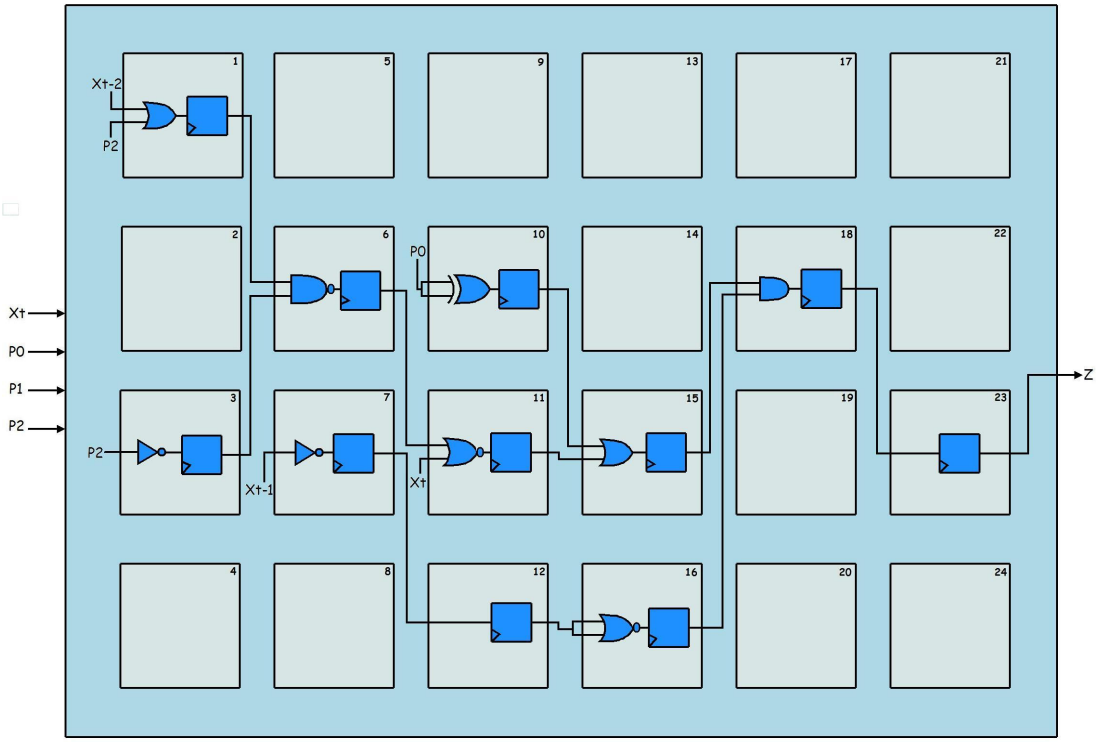

Ilustración 29. Esquema del diseño de un reconocedor del patrón $P_2P_1P_0$=011 evolucionado por nuestro sistema.

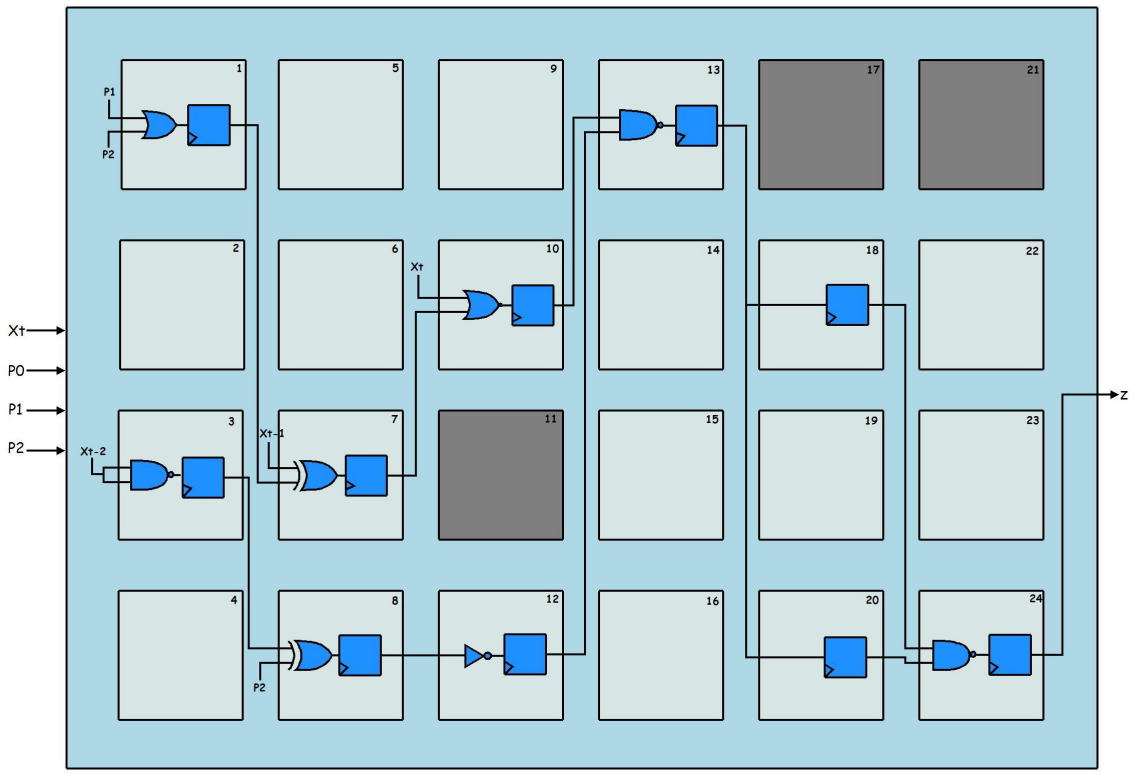

Ilustración 30. Alternativa de diseño para el reconocedor de la Ilustración 29. Tras el reconocimiento de un fallo en tres celdas del circuito virtual, el propio componente se recupera y continúa la búsqueda hasta encontrar una solución compatible con el resto de celdas activas.

La Ilustración 29 muestra la evolución de una solución sin que se haya producido ningún tipo de fallo en el sistema. En este caso hemos modificado los switches de la placa de desarrollo y trabajamos sobre un reconocedor del patrón 011. Por otra parte, la Ilustración 30 muestra la evolución de un sistema en el que tras un uso intensivo (1000 generaciones), se producen diversos fallos en tres de las celdas y por ello concluye evolucionando una solución diferente, que no cuenta con ninguna de los tres módulos rotos.

Conviene incidir en el hecho de que el número de celdas muertas y el tiempo de evolución de un circuito resultan proporcionales. Esto se debe a que cada vez que una celda no puede ser utilizada como parte del circuito, las restricciones para desarrollar una solución correcta aumentan de forma considerable.

5. CONCLUSIONES Y TRABAJO FUTURO

Mediante este trabajo de investigación hemos querido estudiar de primera mano el Hardware Evolutivo, un novedoso enfoque en la percepción del desarrollo de circuitos integrados que se caracteriza por surgir de la unión de la computación evolutiva y los dispositivos de lógica reconfigurable. Éste puede ir dirigido a la elaboración de soluciones concretas estáticas, donde lo que queremos es desarrollar un sistema que tenga un comportamiento específico. O también puede ir dirigido a la elaboración de sistema auto-adaptativos, que buscan dinámicamente soluciones aún siendo influidos por cambios en el entorno.

En ese sentido, y tras un estudio profundo de los orígenes, teorías y tipos de sistemas diseñados hasta ahora, podemos concluir primeramente que este interesante campo de investigación tiene un potencial enorme en un futuro a corto y medio plazo. El hecho de que sea una materia todavía muy inexplorada la hace de por sí atractiva, pero sobre todo, lo que la hace interesante y potencialmente útil es su naturaleza. El hecho de poder llevar a cabo desarrollos de manera automática, simplemente mediante la definición de comportamientos (el cómo), puede

revolucionar el mundo del desarrollo hardware. Por supuesto, ciertos problemas deben ser resueltos, sin los cuales este enfoque resulta poco eficiente. El más importante de ellos es el de la escalabilidad, pues para sistemas de cierto tamaño, este enfoque resulta inabordable.

Respecto al problema de la escalabilidad es necesario enfatizar que estos diseños resultan abordables hasta cierto punto. Durante la realización de este trabajo de investigación hemos podido desarrollar un reconocedor de patrones de tres bits, un sistema cuya funcionalidad no es muy complicada. Pero es que hasta el momento, y como adelantábamos en la primera parte de este documento, los desarrollos llevados a cabo han sido sistemas sin demasiada complejidad. Entre estos se encuentran sumadores de pocos bits, multiplicadores de dos o tres bits, filtros de edición de imágenes, etc. Es por eso que debemos concluir que con las técnicas actuales no resulta ni mucho menos abordable la evolución de, por ejemplo, un microprocesador de propósito general.

Más interesantes resultan, si cabe, los sistemas de hardware evolutivo auto-adaptativo, los cuales pueden someterse a ciertas condiciones o variaciones que se producen en el medio ambiente que les rodea. Supongamos un satélite en el espacio exterior que bajo ciertas circunstancias se sobrecaliente y parte de su lógica pierda su utilidad. En sistemas convencionales, este problema sería trágico y probablemente conlleve la pérdida del mismo. Con el enfoque aquí expuesto, mientras el núcleo evolutivo quede a salvo y se disponga de lógica reconfigurable no dañada en cantidad suficiente, el propio proceso evolutivo detectará esos fallos y buscará una nueva solución enfocándose en las partes activas.

En cuanto al hardware reconfigurable, otro de los aspectos fundamentales de este campo, hemos de admitir que las soluciones existentes en la actualidad son aptas para implementar sistemas de este estilo. Teniendo en cuenta el tamaño reducido de los diseños, la cantidad de lógica contenida no supondrá una restricción grave. Siempre podríamos, además, contar con la posibilidad de unir varios dispositivos aumentando con ello el tamaño. Lo que si supone una restricción considerable es el modo de reconfigurar el sistema. Bien es cierto que se han seguido muchos enfoques para su implantación, como los JBits [12] o los VRC [14], [15], [16].

Puesto que este proyecto pretende ser un primer paso en el hardware evolutivo, las ampliaciones y mejoras funcionales que podríamos añadirle son infinitas. Más aún teniendo en cuenta que se trata de un campo inexplorado, con lo que las esas nuevas características quedan restringidas solamente por nuestra imaginación.

Tras todo diseño e implementación, siempre hay ciertos puntos del propio sistema con los que no quedas completamente satisfecho, pero que son soluciones que se han tenido que tomar de cara a la finalización completa de mismo en el plazo prefijado. Es por eso que las primeras mejoras que proponemos al sistema son las enfocadas a la mejora nuestra propia solución.

La placa de desarrollo con la que contamos contiene dos instancias del procesador para diseños empotrados PowerPC 405. Resultaría muy interesante y

creemos que no muy complicado, pues las interfaces de comunicación y bus de intercambio son similares, modificar el diseño de manera que incluya este procesador en vez del MicroBlaze sintetizado. Según nuestras estimaciones, debería producirse una mejora en los tiempos de ejecución, pues el rendimiento del primer procesador supera con creces al segundo.

También creemos que sería muy interesante poder mejorar la función de coste para la solución implementada. Por su naturaleza, un reconocedor de patrones usual devuelve un valor nulo para todas las combinaciones de entrada, excepto para la correcta. Teniendo en cuenta que para una función aleatoria es sencillo devolver un valor nulo en todas las ocasiones, y que la función de coste implementada se basa únicamente en comparar salidas obtenidas y esperadas, es fácil que los primeros circuitos evolucionados alcancen rápidamente un valor alto del rengo del fitness. Para mejorarlo resultaría necesario incluir más combinaciones de entrada sobre las que trabajar y cuantificar aciertos, y la inclusión de nuevos parámetros de calificación. Aunque no sea lo más deseable por la pérdida de portabilidad, podrían añadirse minimizaciones de las medidas de área consumida y consumo de potencia, con lo que además podríamos dirigir la búsqueda hacia soluciones óptimas. Una comparación de resultados sería intresante.

Tanto los autores de los trabajos en los que basamos nuestra arquitectura interna, como nosotros mismos, proclamamos que el enfoque de los VRC resulta genérico y que con unas mínimas modificaciones es posible implementar otro tipo de sistemas. Resultaría interesante ver si esto es posible para otros diseños secuenciales, como por ejemplo un controlador caracterizado por algún autómata sencillo.

Una de las características más interesante del sistema es la que detecta y recupera al mismo de un fallo inesperado. Esta rama es la que probablemente pueda proveer mejores resultados de investigación futuros, por lo que sería recomendable exprimir todas sus posibilidades. Teniendo en cuenta que los dispositivos reconfigurables cuentan con lógica suficiente para ello, y nuestra arquitectura permite la reconfiguración dinámica y rápida, sólo es necesario implementar un modelo que permita dicha recuperación. Partiendo de que esto ya ha sido implementado, el futuro investigador podría enfocar su investigación hacia la recuperación ante diferentes tipos de fallos o a la reinicialización del modelo en caso de que ya no se produzcan. También sería muy interesante intentar incluir en esa reconfiguración una restricción de carácter temporal, de forma que se asegure al usuario que su sistema será capaz de detectar y recuperarse de un fallo en un tiempo máximo predefinido.

Como podemos imaginar tras la reflexión en estas últimas líneas, las posibilidades dentro de este campo tan inexplorado como atractivo son enormes, y su futuro, prometedor.

6. BIBLIOGRAFÍA

[1] deGaris, H. Evolvable hardware – genetic programming of a Darwin. *International conference on Artificial Neural Netwroks and Genetic algorithms.* Innsbruck, Austria. Springer-Verlag, 1993.

[2] Higuchi, T. et al. Evolvable hardware: A first step towards building a Darwin Machine. *In Proceedings of the 2nd International Conference on Simulated Behaviour.* Pages 417-424. MIT Press, 1993.

[3] Torresen, J. An Evolvable Hardware Tutorial. 2003.

[4] Greenwood, T. and Bentley, P. On Evolvable Hardware.

[5] Greenwood, T. and Tyrrell, A. Introduction to Evolvable Hardware: A Practical Guide for Designing Adaptive Systems. *IEEE Press Series on Computational Intelligence 2007.*

[6] H de Garis. LSL evolvable hardware workshop report. ATR Japan. Technical Report, 1995.

[7] Higuchi, T., Liu, Y. and Yao, X. Evolvable Hardware. *Genetic and Evolutionary Computation Series*. Springer, 2006.

[8] Yao, X. and Higuchi, T. Promises and challenges of evolvable hw. *IEEE Transactions on systems man and cybernetics-part c: applications and reviews*. Vol. 29, No. 1, February 1999.

[9] Higuchi, T. Evolvable hw with genetic learning. Procedures of Simulated Adaptive Behavior. 417-424. MIT Press. 1993.

[10] Murakawa, M. et al. Evolvable hardware at function level. *Proceedings of the Parallel Problem solving from nature conference. V*ol. 1141, pp.62-71. 1996.

[11] Thompson, A. Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution. Springer, Berlin, UK, 1998.

[12] Torresen, J. A divide a conquer approach to evolvable hardware. *Proceeding of the 2nd International conference on evolvable Systems: from biology to hardware.* Vol. 1478, pp57-65. 1998.

[13] Torrensen, J. A scalable approach to evolvable hardware. *Genetic programming and evolvable machines*. 3(3) 259-282. 2002.

[14] Sekanina, L. Evolvable Components: from theory to hardware implementations. *Natural Computing Series*. Springer Verlag, 2004.

[15] Sekanina, L. Virtual Reconfigurable Circuits for Real-World Applications of Evolvable Hardware. *ICES 2003.* LNCS 2606, pp. 186–197, 2003. Springer-Verlag. Berlin Heidelberg, 2003.

[16] Sekanina, L. Towards Evolvable IP Cores for FPGAs. *Proceedings of the 2003 NASA/Dod Conference on Evolvable Hardware.* IEEE, 2003.

[17] Vašícek, Z. and Sekanina, L. An evolvable hardware system in Xilinx Virtex II Pro FPGA. *Int. J. Innovative Computing and Applications*, Vol. 1, No. 1, 2007.

[18] K. Vassilev, V., Job, D. and F. Miller, J. Towards the Automatic Design of More Efficient Digital Circuits. IEEE, 2000.

[19] Vassilev, V., Miller, J. et al. On the nature of two-bit multiplierlandscapes. *Proceeding of the 1st NASA/DoD Workshop on evolvable hardware.*1999.

[20] Holland, J. Adaptation in Natural and Artificial Systems. Ann Arbor, MI: University of Michigan Press, 1975.

[21] D.Lohn, J., S. Hornby, G. and S. Linden, D. An Evolved Antenna for Deployment on NASA's Space Technology 5 Mission.

[22] Tufte, G. and C. Haddow, P. Evolving and Adaptive Digital Filter. IEEE, 2000.

[23] J.Hirst, A. Notes on evolution of adaptive hardware. *2nd International Conference of Adaptive Computer Engineering Design* (ACEDC'96).

[24] Nirmalkumar, P., Raja, J., Perinbam, P., Ravi, S. and Rajan, B. On Suitability of FPGA Based Evolvable Hardware Systems to Integrate Reconfigurable Circuits with Host Processing Unit. *IJCSNS International Journal of Computer Science and Network Security*, VOL.216 6 No.9A, September 2006.

[25] Greenwood, T., Ramdsen, E. and Ahmed, S. An empirical comparison of evolultionary algorithms for evolvable hardware with maximum time to reconfigure requirements. *Proceedings 2003 NASA/DoD conference on evolvable hardware.* 59-66.2003.

[26] Greenwood, T. On the practicality of using intrinsic reconfigurations for fault recovery. *IEEE Transactions on Evolutionary Computation.* 9(4) 398-405. 2005

[27] Dianati, M., Song, I. and Treiber, M. An Introduction to Genetic Algorithms and Evolution Strategies.

[28] Slany, K. Branch Predictor On-Line Evolutionary System.

[29] Sekanina, L., Martinek, T., and Gajda, Z. Extrinsic and Intrinsic Evolution of Multifunctional Combinational Modules. *IEEE Congress on Evolutionary Computation*. 2006, Canada.

[30] Sekanina, L. and Drabeck, V. Relations between fault tolerance and reconfigurations in cellular systems. *Proceedings of the 6th IEEE online testing workshop*. 25-30.2000

[31] Glette, K. and Torresen, J. A Flexible On-Chip Evolution System Implemented on a Xilinx Virtex-II Pro Device. *In Proceedings of ICES*. 2005, pp.66-75.

[32] Zebulum, R., A. Pacheco, M. and Vellasco, M. Analog circuit evolution in extrinsic and intrinsic modes. *Proceedings of the 2nd International Conference on Evolvable Systems: From Biology to Hardware*, volume 1478 of Lecture Notes in Computer Science, pages 154–165. Springer-Verlag. Berlin-

Heidelberg, 1998.

[33] Keymeulen, D., Zebulum, R. and Stoica, J. Fault tolerant evolvable hardware using field programmable transistors arrays. *IEEE Transactions on Realiability.* 49(3) 113-122. 2000

[34] Vigraham, Saranyan A. An Analog Evolvable Hardware Device for Active Control. 2009

[35] F. Miller, J. and Thomson, P. Cartesian Genetic Programming. *Proceedings of the 3rd European Conference on Genetic Programming (EuroGP2000)*. Volume 1802 of *Lecture Notes in Computer Science*, pages 121–132. Springer-Verlag. Berlin, 2000.

[36] F. Miller, J., Job, D. and K. Vassilev, V. Principles in the Evolutionary Design of Digital Circuits—Part I. *Genetic Programming and Evolvable Machines*, 1, 7-35. Kluwer Academic Publishers, 2000.

[37] F. Miller, J., Job, D. and K. Vassilev, V. Principles in the Evolutionary Design of Digital Circuits—Part II. *Genetic Programming and Evolvable Machines*, 1(2), 259-288. Kluwer Academic Publishers, 2000.

[38] Banzhaf, W. Et al. Genetic Programming: An Introduction. *Morgan Kaufmann Series in Artificial Intelligence.* 1998.

[39] R. Koza, J. Genetic Programming. Cambridge, MA: MIT Press, 1992.

[40] R. Koza, J. Genetic Programming II. Cambridge, MA: MIT Press, 1994.

[41] R. Koza, J., H. Bennett, F., Andre, D. And A. Keane, M. Four problems for which a computer program evolved by genetic programming is competitive with human performance. *Proceedings of the 1996 IEEE International Conference in Evolutionary Computation (ICEC'96)*. Piscataway, NJ: IEEE, pp. 1–10.

[42] R. Koza, J., H. Bennett, F. Andre, D. A. Keane, M. and Dunlap, F. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions in Evolutionary Computing.* vol. 1, pp. 109–128. February, 1997.

[43] Whigham, P. Grammatically-based genetic programming. *In Proceedings of the Workshop in Genetic Programming: From Theory to Real-World Applications.* New York: Morgan Kaufmann, July 1995, pp. 33–41

[44] Michalewicz, Z. Genetic Algorithms + Data Structures = Evolution Programs. 3rd, rev. and extended ed. 1996. Corr. 2nd printing, 1998, XX, 387 p. 68 illus.,

Hardcover.

[45] Whigham, P. Inductive bias and genetic programming. *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, (GALESIA).* pp. 461–466. Sept. 1995.

[46] Whigham, P. and McKay, R. Genetic approaches to learning recursive relations. *Progress in Evolutionary Computation*. vol. 956. pp. 17–27. Springer-Verlag. Berlin- Heidelberg, 1995.

[47] Whigham, P. A schema theorem for context-free grammars. *Proceedings of the 1995 IEEE International Conference in Evolutionary Computation (ICEC'95)*. Piscataway, NJ: IEEE Press, vol. 1, pp. 178–182.

[48] M. Kling, R. and Banerjee, P. ESP: Placement by simulated evolution. *IEEE Transactions. Computer-Aided Design*. vol. 8, pp. 245–256. March, 1989.

[49] Hemmi, H., Mizoguchi, J. and Shimohara, K. Development and evolution of hardware behaviors. *In Artificial Life IV: Proceedings of the 4th International Workshop Synthesis Simulation Living Systems.* Cambridge, MA: MIT Press. pp. 371–376. 1994.

[50] Li, W. et al. Wetware as a bridge between computer engineering and biology. *In Preliminary Proceedings of the 2nd European Conference on Artificial Life*. (ECAL'93). Pg 658-667. 1993.

[51] Perotto, J. F. et al. Life in silico. *11th European Conference on Circuit Theory and Design.* (ECCTD'93) 145-149. 1993.

[52] P. Fourman, M. Compaction of symbolic layout using genetic algorithms. *In Proceedings of the 1st International Conference in Genetic Algorithms and Their Application*. Ed. Pittsburgh, PA: Carnegie Mellon Univ. Press. 1985, pp. 141–153.

[53] Cohoon, J. and Paris, W. Genetic placement. *In Proceedings of the International Conference of Computer-Aided Design*. New York, IEEE Press. pp. 422–425. 1986.

[54] Mange, D., Sipper, M., Stauffer, A. and Tempesti, G.. Embryonics: A new methodology for designing field programmable gate arrays with self-repair and self-replicating properties. *Proceedings of the IEEE.* 88(4) 416-541. 2000.

[55] Tyrrel, A., Hollingworth G. Evolutionary strategies and intrinsic fault tolerance. Proceedings of the 3rd NASA/DoD Workshop on Evolvable Hw 98-106. 2001.

[56]Tyrrell, A., Sánchez, M., Floreano et al. POEtic tissue: an integrated architecture for bio inspired hw. *Proceedings of the 5th International Conference on Evolvable Systems.*129-140. 2003.

[57]Thompson, A. On the automatic design of robust electronics through artificial evolution. *Proceedings of the 2nd International Conference on Evolvable Systems: from biology to hardware. V*ol. 1478, pp.13-35. 1998.

[58]Thompshon, A. Layzell, P. and Zebulum R.S. Explorations in design space: unconventional electronics design through artificial evolution. *IEEE Transactions on Evolutionary Computation.* 3(3)167-196.1999.

[59]Stoica, A,, Zebulum, R. and Keymeulen, D. Mixtrinsic evolution. *ICES '00. Proceedings of the 3rd International Conference on evolvable systems: from biology to hardware.*. Vol. 1801, pp 208-217. 2000.

[60]Saraeel, M., Nikoofar, H. and Manzour, A. Epistasy Search in Population-Based Gene Mapping Using Mutual Information. *IEEE International Symposium on Signal Processing and Information Technology*. 2007.

[61]Xilinx Inc. ISE 10.1 Quick Start Tutorial. 2008.

[62]Xilinx Inc. ISE 10 Help: http://www.xilinx.com/itp/xilinx10/isehelp/isehelp_start.htm

[63]Xilinx Inc. MicroBlaze Processor Reference Guide: Embedded Development Kit EDK 10.1i. v9.0, 2008.

[64]Xilinx Inc. EDK Concepts, Tools and Techniques: A hands-on guide to Effective Embedded System Design (v 9.1i). 2007.

[65]Xilinx Inc. Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet. V4.7, 2007.

[66]Xilinx Inc. Xilinx University Program Virtex-II Pro Development System: Hardware Reference Manual. v1.1, 2008.

[67]Xilinx Inc. LibXil Standard C Libraries (Software libraries available for embedded processors). EDK 9.1i, 2006.

[68]Portal del Institute of Electrical and Electronics Engineers: http://www.ieee.org/portal/site

-APÉNDICE-

COMPLICACIONES Y PROBLEMAS SUPERADOS

Como ya comentábamos al principio de esta memoria, el Hardware Evolutivo es un campo de investigación bastante reciente. Puesto que este trabajo de investigación ha resultado ser un primer paso en la materia, con él se pretende tener una aproximación práctica, real y completa a ella. Al mismo tiempo, se pretende tener continuidad en el tiempo y seguir en un futuro cercano explorando sus amplias posibilidades. Hemos creído conveniente y muy útil para el futuro investigador que continúe dicha senda la inclusión de un apéndice que explique de la manera más fehaciente posible las principales complicaciones con las que nos hemos enfrentado, y hemos superado de una u otra manera, durante la fase de diseño e implementación.

El Hardware Evolutivo se sustenta sobre la ingeniería de computadores, el hardware reconfigurable, la inteligencia artificial y los algoritmos evolutivos. Es por ello, que se recomienda el conocimiento lo más amplio posible de estas materias, sus técnicas y teorías, de cara a abordar un diseño de este estilo. Entrando más en detalle, y ya que los dispositivos sobre los que se trabaja son las FPGA, un amplio conocimiento de éstas, su funcionamiento y programación serán más que bienvenidos. El estar familiarizados con el lenguaje de modelado hardware VHDL (o Verilog), con el lenguaje de programación C (C++), con las herramientas EDK e ISE de Xilinx y con el simulador ModelSim de Mentor Graphics resulta un punto de partida idóneo para penetrar en este mundo.

A continuación, se enumeran las principales complicaciones de diseño, utilización y demás que hemos solventado durante el desarrollo:

- Bien es cierto que no somos expertos en la utilización de la herramienta de simulación temporal de circuitos ModelSim, pero tampoco hasta ahora hemos encontrado respuesta al por qué de un comportamiento como el descrito a continuación cuando la opción de optimización de dicha aplicación se encuentra activada. El fichero ModelSim.ini es su archivo de configuración, y constituye por sí mismo un listado de las características y opciones que implementará al utilizarlo. En él se encuentra, activada por defecto, una opción de optimización para la simulación de circuitos denominada vopt, y que a pesar de su nombre no nos dio más que quebraderos de cabeza durante las primeras fases de desarrollo de los sistemas. Y es que suprimía alguna de las señales de los módulos internos sin razón aparente, aún siendo útiles para otros y aún teniendo un valor estable.

Aconsejamos desactivar esta opción y así evitar potenciales momentos de histeria y la correspondiente pérdida de tiempo.

- Bien es cierto que los diseños hardware de cierto nivel de dificultad resultan complicados. Más si cabe resultan los sistemas que aúnan partes software y hardware, como pueden ser los System-on-Chip o los co-diseños HW-SW. Este enfoque nos ha traído numerosos quebraderos de cabeza, pues su implementación, a pesar de contar con herramientas tan potentes como EDK o ISE, resulta bastante engorrosa. Es sobre todo la comunicación entre el IP y el procesador lo que conlleva más problemas. Así mismo, la simulación de dichos sistemas bajo ModelSim resulta muy complicada puesto que se alarga mucho en el tiempo. La inclusión de un procesador hace que en la simulación temporal aparezcan cientos de instrucciones dedicadas al tratamiento de rutinas que nada tienen que ver con la ejecución de nuestra aplicación. Mención aparte requieren las instrucciones de impresión tipo printf para mostrar información por el HyperTerminal de nuestro PC. Aconsejamos suprimirlas al completo, pues el envío de dicha información introduce muchísimos ciclos inútiles en cuanto a la depuración del sistema.

- En este mismo ámbito se encuentra otro de los problemas a los que nos enfrentamos durante la simulación del sistema al completo. Nuestro diseño inicial incorporaba uno de los Power PC 405 incluidos en la FPGA Virte-II Pro. Sin embargo, al ser el sistema operativo que utilizamos en nuestro PC Windows Vista, los errores han sido mayúsculos, de hecho no pudimos llegar a simularlo. Es necesario incluir y compilar las bibliotecas de simulación de EDK para ModelSim en caso de querer visualizar las señales de un procesador en una simulación temporal, pero en nuestro caso, que es además un error admitido de Xilinx, dicha ejecución no era posible. Aún siguiendo las instrucciones de resolución que la propia empresa proporcionaba como solución, no fue posible resolver el problema. De hecho, este percance nos obligo a cambiar el diseño y optar por el soft-core sintetizado MicroBlaze, el cual no dio problemas en este sentido. En caso de querer utilizar Windows Vista y ModelSim y simular un PowerPC 405, se recomienda de antemano visitar las páginas de documentación que Xilinx tiene abiertas al respecto.

- El HyperTerminal ha resultado ser una herramienta de vital importancia en nuestro desarrollo pues de manera sencilla permite visualizar información y datos de la ejecución del algoritmo que se ejecuta en el procesador. Para ello simplemente se necesita incluir una unidad UART al sistema, conectarla convenientemente al bus PLB y elegir sus puertos como entradas y salidas estándar del procesador. Sin embargo, puesto que se trabaja a un nivel diferente, no es nunca posible visualizar señales que se transmiten entre los diferentes módulos de nuestro IP. Para llevar a cabo esto último es necesario realizar simulaciones largas o incluir ciertos registros que apunten a posiciones de memoria que después podamos visualizar ya sea imprimiéndolas por pantalla o mediante una herramienta como XMD.

- La primera aproximación al Circuito Virtual Reconfigurable que seguimos fue la misma que Sekanina en sus publicaciones, de las cuales la más característica [14]. Esta se enfoca directamente a los circuitos combinacionales. El hecho de que se permitan saltos entre las celdas reconfigurables internas es una opción que incluimos en un primer momento. Lo que no tuvimos en cuenta es que la naturaleza de nuestro sistema es secuencial, y por ello incluye biestables a la salida de cada celda y se ejecuta en un cierto número de ciclos concreto. Al permitir estos saltos entre celdas, resultaba imposible determinar cuantos ciclos realmente tarda en computar cada una de las combinaciones de entrada un circuito, y por ello resultaba sumamente complicado validar su diseño interno. Como solución se suprimieron esos saltos, pero se permitió que las columnas segunda y tercera quedaran conectadas a las entradas totales del VRC además de a las salidas de la columna inmediatamente anterior. Con esto obligamos a que el retardo de cálculo de la función interna sean 6 ciclos (debido a las 6 etapas de celdas con biestables), y con ello la validación de salidas no resulta un proceso aleatorio.

- Uno de los problemas que más consecuencias ha tenido y tiene en nuestro diseño final es el relativo a la función de coste, aptitud o fitness del circuito elegido. La naturaleza de los reconocedores de patrones hace que su salida se active sólo cuando la cadena correspondiente aparece en la entrada. En este caso se trata de 3 bits, por lo que trabajamos sobre 8 posibles combinaciones de entrada. Por una parte sabemos que las otras 7 ternas darán una solución nula a la salida del sistema, y teniendo en cuenta las funciones implementadas en las celdas funcionales, resulta sencillo diseñar circuitos que devuelvan, al menos esos siete ceros. Por otra parte, y como ya explicamos más arriba, la función de fitness se define con un rango de 0 a 8, siendo 0 cuando todas las salidas son invertidas respecto a las esperadas, y 8 cuando las salidas son exactamente iguales. Al sumar ambas características obtenemos que el rango de la función, aparte de ser muy pequeño y no permitir que el algoritmo evolutivo desarrolle todo su potencial, implica que desde una primera generación encontraremos soluciones con valoración 7, correspondiente al reconocimiento de todos los ceros. Hemos logrado restar importancia al problema aumentando la entrada serie del sistema, hasta los 26 bits, con lo que logramos una función de fitness que se mueve entre 0 y 24. En cualquier caso, y extrapolando los valores, el inconveniente sigue estando latente. Otra posible solución consiste en mejorar la función de aptitud mediante la inclusión de otras restricciones de área, consumo, etc. al sistema, pero eso iría en contra de lo planteado en un principio como propuestas ya que atenta contra la portabilidad de sistema.

- De cara a llevar a cabo comparaciones entre las ejecuciones parametrizadas del sistema, pensamos en la inclusión de un temporizador como medidor de tiempos. Sin embargo, puesto que esos retardos van a ser muy variables, e incluso van a llegar a durar minutos, esta solución ha quedado descartada. El susodicho temporizador esta incluido en las librerías de IP del EDK como un registro de 32bits. Por ello, con él seremos capaces de contar hasta 2^32 ciclos de reloj. Teniendo en cuenta que nuestro sistema trabaja a una frecuencia de

100MHz, el resultado deberá concluir en menos de 43 segundos para que la cuenta sea efectiva. En caso contrario el contador se saturará. Como solución decidimos llevar a cabo las mediciones de tiempo exteriormente.

- Algo que puede parecer a simple vista una nimiedad puede llegar a ser un problema engorroso a la hora de testear el sistema. En el desarrollo suponemos un operador del sistema que decide en que momento comienza la evolución, en caso de querer reinicializar el sistema, cuando se lleva a cabo… todo ello mediante la utilización de los botones de la placa. Además, decide el patrón que el sistema deberá ser capaz de reconocer mediante los switches que también incorpora. Aunque parezca raro no es clara su identificación debido a la confusión que crean las dos serigrafías que lleva impresas, y a que es un componente que se activa a baja, es decir, su estado ON devuelve un 0 lógico por sus salidas. Un esquema gráfico con la correspondencia a los bits del patrón de entrada al sistema se muestra en la Ilustración 31.

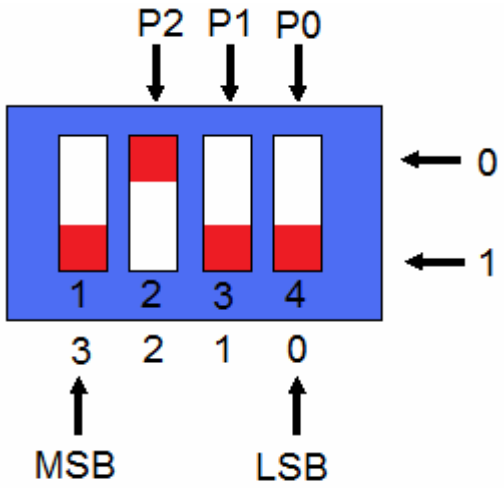

Ilustración 31. Esquema gráfico de la disposición de los switches en la placa XUPV2P.